CoreMedia Content Cloud

Frontend Developer Manual

CoreMedia

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

December 17, 2024 (Release 2412.0)

List of Figures

List of Tables

List of Examples

1. Preface

This manual describes frontend development tasks in *CoreMedia Content Cloud*.

- Chapter 2, *Quick Start* [14] describes the prerequisites for the frontend development, how to set up the development environment and the structure of the workspace.
- Chapter 3, *Web Development Workflow* [19] describes the Frontend Development Workflow.
- Chapter 4, *Workspace Concept* [29] describes the concept and structure of the workspace, the themes and bricks.
- Chapter 6, *Reference* [102] describes all available themes, bricks and APIs.

1.1 Audience

This manual is intended for frontend developers who plan to develop a frontend for the CoreMedia system.

1.2 Typographic Conventions

CoreMedia uses different fonts and types in order to label different elements. The following table lists typographic conventions for this documentation:

Element	Typographic format	Example
Source code Command line entries Parameter and values Class and method names Packages and modules	Courier new	`cm systeminfo start`
Menu names and entries	Bold, linked with |	Open the menu entry **Format|Normal**
Field names CoreMedia Components Applications	Italic	Enter in the field *Heading* The *CoreMedia Component* Use *Chef*
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is *not* saved
Buttons	Bold, with square brackets	Click on the **[OK]** button
Code lines in code examples which continue in the next line	\	`cm systeminfo \` `-u user`

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

Pictograph	Description
	Tip: This denotes a best practice or a recommendation.
	Warning: Please pay special attention to the text.
	Danger: The violation of these rules causes severe damage.

Table 1.2. Pictographs

1.3 CoreMedia Services

This section describes the CoreMedia services that support you in running a CoreMedia system successfully. You will find all the URLs that guide you to the right places. For most of the services you need a CoreMedia account. See Section 1.3.1, “Registration” [5] for details on how to register.

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

Find the latest overview of all CoreMedia services and further references at:

http://documentation.coremedia.com/new-user-orientation

- Section 1.3.1, “Registration” [5] describes how to register for the usage of the services.
- Section 1.3.2, “CoreMedia Releases” [6] describes where to find the download of the software.
- Section 1.3.3, “Documentation” [7] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.3.4, “CoreMedia Training” [10] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.3.5, “CoreMedia Support” [10] describes the CoreMedia support.

1.3.1 Registration

In order to use CoreMedia services you need to register. Please, start your initial registration via the CoreMedia website. Afterwards, contact the CoreMedia Support (see Section 1.3.5, “CoreMedia Support” [10]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

CoreMedia provides its software as a Maven based workspace. You can download the current workspace or older releases via the following URL:

https://releases.coremedia.com/cmcc-12

Refer to our Blueprint Github mirror repository for recommendations to upgrade the workspace either via Git or patch files.

NOTE

If you encounter a 404 error then you are probably not logged in at GitHub or do not have sufficient permissions yet. See Section 1.3.1, “Registration” [5] for details about the registration process. If the problems persist, try clearing your browser cache and cookies.

Maven artifacts

CoreMedia provides parts of its release artifacts via Maven under the following URL:

https://repository.coremedia.com

You have to add your CoreMedia credentials to your Maven settings file as described in section Section 3.1, “Prerequisites” in *Blueprint Developer Manual* .

npm packages

CoreMedia provides parts of its release artifacts as npm packages under the following URL:

https://npm.coremedia.io

Your pnpm client first needs to be logged in to be able to utilize the registry (see Section 3.1, “Prerequisites” in *Blueprint Developer Manual*).

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.3.5, “CoreMedia Support” [10]) to get your licences.

1.3.3 Documentation

CoreMedia provides extensive manuals, how-tos and Javadoc as PDF files and as online documentation at the following URL:

https://documentation.coremedia.com

The manuals have the following content and use cases:

Manual	Audience	Content
Adaptive Personalization Manual	Developers, architects, administrators	This manual describes the configuration of and development with *Adaptive Personalization*, the CoreMedia module for personalized websites. You will learn how to configure the GUI used in *CoreMedia Studio*, how to use predefined contexts and how to develop your own extensions.
Analytics Connectors Manual	Developers, architects, administrators	This manual describes how you can connect your CoreMedia website with external analytic services, such as Google Analytics.
Blueprint Developer Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of *CoreMedia Content Cloud*. It describes the content type model, the *Studio* extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features. It also describes the concepts and usage of the project workspace in which you develop your CoreMedia extensions. You will find a description of the Maven structure, the virtualization concept, learn how to perform a release and many more.
Connector Manuals	Developers, administrators	This manuals gives an overview over the use cases of the eCommerce integration. It describes the deployment of the Commerce Connector and how to connect it with the CoreMedia and eCommerce system.
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.

Manual	Audience	Content
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an *Importer* application and how you define the transformations that convert your content into CoreMedia content.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different otions to desgin your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.

Manual	Audience	Content
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.
Utilized Open Source Software & 3rd Party Licenses	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

Table 1.3. CoreMedia manuals

If you have comments or questions about CoreMedia's manuals, contact the Documentation department:

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

CoreMedia's training department provides you with the training for your CoreMedia projects either live online, in the CoreMedia training center or at your own location.

You will find information about the CoreMedia training program, the training schedule and the CoreMedia certification program at the following URL:

http://www.coremedia.com/training

Contact the training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

CoreMedia's support is located in Hamburg and accepts your support requests between 9 am and 6 pm MET. If you have subscribed to 24/7 support, you can always reach the support using the phone number provided to you.

To submit a support ticket, track your submitted tickets or receive access to our forums visit the CoreMedia Online Support at:

http://support.coremedia.com/

Do not forget to request further access via email after your initial registration as described in Section 1.3.1, “Registration” [5]. The support email address is:

Email: support@coremedia.com

Create a support request

Support request

CoreMedia systems are distributed systems that have a rather complex structure. This includes, for example, databases, hardware, operating systems, drivers, virtual machines, class libraries and customized code in many different combinations. That's why CoreMedia needs detailed information about the environment for a support case. In order to track down your problem, provide the following information:

- Which CoreMedia component(s) did the problem occur with (include the release number)?
- Which database is in use (version, drivers)?
- Which operating system(s) is/are in use?
- Which Java environment is in use?
- Which customizations have been implemented?
- A full description of the problem (as detailed as possible)
- Can the error be reproduced? If yes, give a description please.
- How are the security settings (firewall)?

In addition, log files are the most valuable source of information.

Support checklist

To put it in a nutshell, CoreMedia needs:

1. a person in charge (ideally, the CoreMedia system administrator)
2. extensive and sufficient system specifications
3. detailed error description
4. log files for the affected component(s)
5. if required, system files

Log files

An essential feature for the CoreMedia system administration is the output log of Java processes and CoreMedia components. They're often the only source of information for error tracking and solving. All protocolling services should run at the highest log level that is possible in the system context. For a fast breakdown, you should be logging at debug level. See Section 4.7, "Logging" in *Operations Basics* for details.

Which Log File?

In most cases at least two CoreMedia components are involved in errors: the *Content Server* log files together with the log file from the client. If you know exactly what the problem is, solving the problem becomes much easier.

Where do I Find the Log Files?

By default, application containers only write logs to the console output but can be accessed from the container runtime using the corresponding command-line client.

For the *docker* command-line client, logs can be accessed using the **docker logs** command. For a detailed instruction of how to use the command, see docker logs. Make sure to enable the timestamps using the `--timestamps` flag.

```
docker logs --timestamps <container>
```

For the *kubectl* command-line client in a Kubernetes environment you can use the **kubectl logs** command to access the logs. For a detailed instruction of how to use the command, see kubectl logs. Make sure to enable the timestamps using the `--timestamps` flag.

```
kubectl logs --timestamps <pod>
```

1.4 Changelog

In this chapter you will find a table with all major changes made in this manual.

Section	Version	Description

Table 1.4. Changes

2. Quick Start

Consistent page design with themes

A consistent page design is essential for a professional website. Apart from the HTML structure reflected by the templates, the layout is mainly controlled by web resources, like CSS, JavaScript and templates. CoreMedia uses themes to bundle these files.

Bricks concept

Bricks are reusable frontend modules for your theme. Mostly, they include templates, some styles and JavaScript functions.

Conflicting interests between developing themes and using themes

Developing and using themes, has some conflicting interests. On the one hand, changes of web resources should be immediately effective on your site, so they must be integrated into the caching and invalidation mechanisms of CoreMedia CMS and thus be maintained in the content repository. On the other hand, frontend developers want to work with their favorite familiar tools and have short round-trips to test their changes.

Develop locally but have resources as content

In order to resolve this conflict, CoreMedia offers the Frontend Development Workflow. Here, changes at the local web resources are automatically visible in the preview CAE. Only when a theme is finished, it will be imported into the repository and can be published.

The following sections give a short introduction to get started:

- Section 2.1, “Prerequisites” [16]
- Section 2.2, “Installation” [17]

Quick Overview

Use the following code snippet to get started quickly, if you are familiar with pnpm and modern web development. You don’t need to install or configure tools like *Webpack* or *Babel*. They are preconfigured and hidden so that you can focus on the code.

```
cd <frontend-workspace>
pnpm install

pnpm run create-theme <name>

pnpm install

cd themes/<name>-theme
pnpm run deploy
pnpm start
```

NOTE

Please note, that you will need to type in the Studio URL to your development system and a valid login when running **pnpm start** or **pnpm run deploy** for the first time. In addition to this, the user, used to login, must be member of a developer group and therefore have developer rights in Studio.

To create a minified bundle of the theme, run **pnpm build**.

For CI/CD purposes, there is also a way using Docker to build the themes. It is described in the
<file>frontend/README.adoc</file>
.

For a deep dive into details of our concepts and APIs, read the following chapters Chapter 3, *Web Development Workflow* [19], Chapter 4, *Workspace Concept* [29] and Chapter 6, *Reference* [102].

2.1 Prerequisites

Required Software

The CoreMedia Frontend Workspace provides scripts to install and build (multiple) themes via Node. In addition to this pnpm is used as a package manager. CoreMedia recommends to use the latest LTS version of *Node.js*.

This workspace does not require a Node backend. The Node installation is only required for the tooling.

The following software is required:

- Node.js = 20.x
- pnpm = 9.3.0

2.2 Installation

Preparing the Workspace

Before you can start developing your themes, you need to install the dependent node modules.

Using pnpm

As a frontend developer, you are probably familiar with *Node.js* and *pnpm* or *npm*. *npm* stands for node package manager and is a way to manage dependencies through *Node.js*. Due to the advantages that *pnpm* offers over *npm*, CoreMedia now recommends *pnpm* for the frontend workspace.

The Frontend Workspace is split into `libs`, `config`, `themes` and `target`. Please note, that `config` will not be created until running **pnpm start** for the first time. See Section 4.1, “Structure of the Workspace” [30] for more information.

Running the following script at the root level of the Frontend Workspace will install all necessary tools and dependencies. It will also automatically check for existing themes and will install their dependencies too.

```
pnpm install
```

Running the following script at the root level of the Frontend Workspace will automatically check for existing themes and will build them. Generated themes will be stored in `target/themes` as zip files.

```
pnpm build
```

NOTE

You need a stable internet connection to install the Frontend Workspace. Otherwise, dependencies cannot be downloaded from the *npm registry*. You need at least access to **https://registry.npmjs.org/** and **github.com** in order to build this workspace. Please check Section 3.1, "Prerequisites" in *Blueprint Developer Manual* for more information on how to configure a proxy.

Some of our third-party dependencies (for example, *node-sass*) will attempt to compile binaries that could not be downloaded via **github.com** itself. If you see error messages like `Error: Can't find Python executable "python", you can set the PYTHON env variable.` this might be just an aftereffect because access to **github.com** was blocked. You do not need `Python` or any other compiler to install the frontend workspace.

Consistent Dependency Versions For Installation

pnpm-lock.yaml

Package managers like *pnpm* support Semantic Versioning when resolving dependencies. This means that dependencies can be specified using version ranges and usually the latest available version is installed. Making use of this feature has become common practice for most of the packages provided via these package managers. This includes packages the frontend workspace depends on. One of the intentions is to make upgrading to newer patches or minor versions easier without additional afford.

While in theory this seems to be a good agreement, relying on the assumption that a patch or minor upgrade will never break a running system has been proven wrong. This is why CoreMedia is fixating the used dependency versions to achieve consistent behavior across different installations (and builds) regardless of the time it is performed. So the same result is achieved no matter at which point of time the Frontend Workspace is being installed.

This is supported by *pnpm* without any additional configuration. After each successful installation via `pnpm install` a `pnpm-lock.yaml` file is generated (or updated) containing the used fixed versions. This file is meant to be checked in and should not be removed as otherwise the information will be lost and *pnpm* will generate a new file with different (in most cases the most current) fixed versions.

Our releases also contain a single `pnpm-lock.yaml` file in the root folder of the Frontend Workspace. Do not remove this file as it contains the dependency versions the Frontend Workspace release was tested with - so these versions are the dependency versions CoreMedia actually supports for that release. If the file is updated (for example, if you have added new dependencies) check in the updates to the version control.

3. Web Development Workflow

This section contains the best practice web development workflow of *CoreMedia*. It describes how to adapt your resource files in the CoreMedia workspace with fast turnaround times and how you can deploy the files to the live system later (see Section 5.4.12, "Client Code Delivery" in *Blueprint Developer Manual* for details). It does not cover how to write CSS or JavaScript files or how to configure and use the *CoreMedia CAE*.

Develop local, deploy global

Web development usually takes place in IDEs or some other kind of source code editor. And since development of web resources, aside from minor changes, shouldn't take place in *CoreMedia Studio*, *CoreMedia Blueprint* provides two solutions depending on the location of the CAE (local or remote) to work with resource files in the workspace until the files are ready to be imported into the content repository.

The following sections explain the details of the web developer workflows:

- Section 3.1, "Using a Remote *CAE*" [20]
- Section 3.2, "Using a Local *CAE*" [25]

Quickstart

Use one of the following pnpm commands inside a theme folder for starting a web developer workflow.

`pnpm start`	Start Developer Mode with CAE, configured in `env.json`. Remote is the default.
`pnpm start --remote`	Start Developer Mode using a remote CAE.
`pnpm start --local`	Start Developer Mode using a local CAE.

Table 3.1. Developer workflow commands

3.1 Using a Remote *CAE*

CoreMedia Blueprint provides a simple yet powerful way for developers to work with workspace resources.

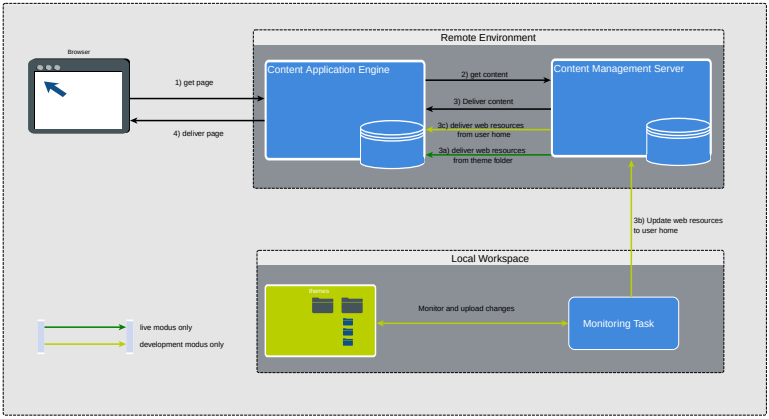

Figure 3.1. CAE flow in detail

Figure 3.1, “CAE flow in detail” [20] gives an overview of the idea behind local resources.

1. The browser requests a page from the remote CAE.
2. The CAE requests the content and web resources from the Content Server.
3. While in development mode, the Content Server delivers web resources from the home folder of the logged in developer.

   In live mode, the Content Server delivers the web resources from the regular source.
4. The CAE combines the content and web resources from the Content Server and delivers the requested page to the browser.

Configuring Studio

The remote web development workflow uses a REST service co-located with Studio for uploading resources. A number of configuration options of the Studio web application control how the REST service operates.

```
themeImporter.themeDeveloperGroups
```

Default	developer

Description	Contains a list of groups whose users are permitted to upload resources (An LDAP group must have the format name@domain). Multiple group names are separated by commas.
`themeImporter.apiKeyStore.basePath`	
Default	`themeImporter/apiKeyStore`.
Description	References the directory in which API keys are stored. This directory must be readable and writable by the Studio application, but should by strongly restricted otherwise, because it contains security relevant data. It is strongly recommended replacing the default relative path with an absolute path.
`themeImporter.apiKeyStore.expiresAfter`	
Default	86400 (1 day)
Description	Defines the number of seconds until an issued API key expires.

Table 3.2. Properties for remote web development workflow REST service

Configuring the Preview CAE

The property `themeImporter.themeDeveloperGroups` of the preview CAE contains the name of the group whose users are permitted to request user-specific pages (An LDAP group must have the format name@domain). Multiple group names are separated by commas.

The property should be configured like for Studio. See Table 3.2, “ Properties for remote web development workflow REST service ” [20]

Editing Source Files

Preview Changes

In general, editing a theme is a straightforward development task. When you edit CSS files, Sass files or JavaScript files, add images and, maybe, write FreeMarker templates you will immediately see all changes in your preview CAE.

All CoreMedia themes provide a `start` script, which starts the monitor mode. This also includes live reloading to automatically reloading your changed files. Immediate preview of your changes only requires a remote preview CAE and running the monitor mode in your theme directory.

Monitoring Changes

Run "pnpm start --remote" for remote development

The monitor mode may be run by executing the command `pnpm start` from your theme directory. The command watches file changes and updates the theme on the remote CAE. To ensure that the theme is up-to-date on the remote CAE, the monitor mode initially provides the current version of the theme to the CAE.

The monitor mode submits file changes using a REST service co-located with Studio. Therefore, it needs an API key which will be generated right after the user has been authenticated. After starting the monitor mode, the API key is being verified. If the verification fails, the user is being prompted to authenticate.

A live reload mode to automatically refresh the browser on file changes is included. The LiveReload server may be configured in an `env.json` file in the config directory of the Frontend Workspace using the options listed below. All CoreMedia Themes are preconfigured and work out of the box. The LiveReload server runs via HTTPS using auto generated certificates per instance.

Option	Type	Default	Description
livereload.host	String	localhost	This defines the host of the live reload server.
livereload.port	Number	35729	This defines the port the live reload server listens on.

Table 3.3. Options to configure live reload server

After the initialization of the monitor mode is completed, it clears the console and displays a hint including the used URL of Studio and Studio preview. The URL of Studio preview or, if not provided, the URL of Studio is being opened in the default browser. Please note that you may need to accept the certificate for the local LiveReload server first by opening the displayed URL in your browser. Otherwise, the live reload mode will not work properly.

NOTE

The file system listeners that automatically rebuild the theme when a file is changed are only active after the initial build has finished. This means that if you change any files during the initial build it will not cause the changes to be detected. Better wait for the console output stating "Webpack is watching the files..." before performing any further changes after starting the monitor mode.

Monitor Mode Behind a Proxy Server

Configuring proxy for monitor mode

In order that the monitor mode still works behind a proxy server, you need to enter the URL of the proxy server when requested during the login of the `theme-importer`. The URL must follow the same rules as mentioned in Section 3.1, "Prerequisites" in *Blueprint Developer Manual* .

> **NOTE**
>
> Many companies use a proxy auto-config (PAC) file which defines how browsers and other user agents choose the appropriate proxy server for fetching a given URL. These files are not supported by the `theme-importer` - neither by pnpm nor Git. As a workaround, you can install a local proxy server which uses a PAC file to decide how to forward a request.

Example

The following example shows the structure of an `env.json` file. The properties `studioUrl` and `previewUrl` will be set automatically when you pass the login of the `theme-importer`.

```
{
  "studioUrl": "https://127.0.0.1/studio",
  "previewUrl": "https://127.0.0.1/preview/servlet/corporate?userVariant=10",

  "proxy": "http://proxy.company.com",
  "monitor": {
    "livereload": {
      "host": "127.0.0.1",
      "port": 9000
    }
  }
}
```

> **NOTE**
>
> If you use custom values for the `livereload` options in the `env.json`, make sure that you customize the LiveReload URL in the corresponding template `Page._developerMode.ftl`.

Quit command

To quit a running monitor mode, press the keys `<Ctrl>+<C>`.

If you want to submit the complete theme at once to the remote CAE, run `pnpm run theme-importer upload-theme`.

Studio Preview

Enable Developer Mode in Studio

To view your changes instantly in the Studio preview, you need to enable the developer mode via the palette icon of the Studio preview. Then the preview Content Application

Engine uses the web resources from the home directory of the logged in developer and generates the preview including your file changes. If the developer mode is enabled, the palette icon is highlighted and a red wrench is displayed in the lower left corner of the preview.

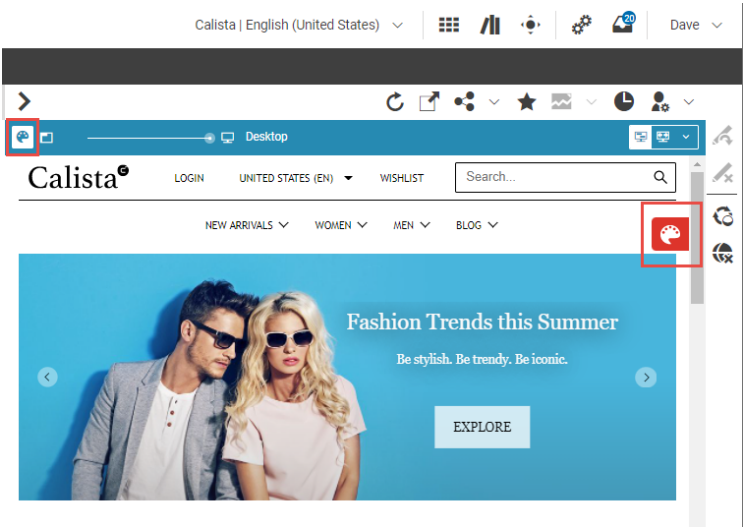

Figure 3.2. Enable Developer Mode in Studio

3.2 Using a Local *CAE*

CoreMedia Blueprint supports local resources as a simple yet powerful way for developers to work with workspace resources, rather than code objects in the content repository.

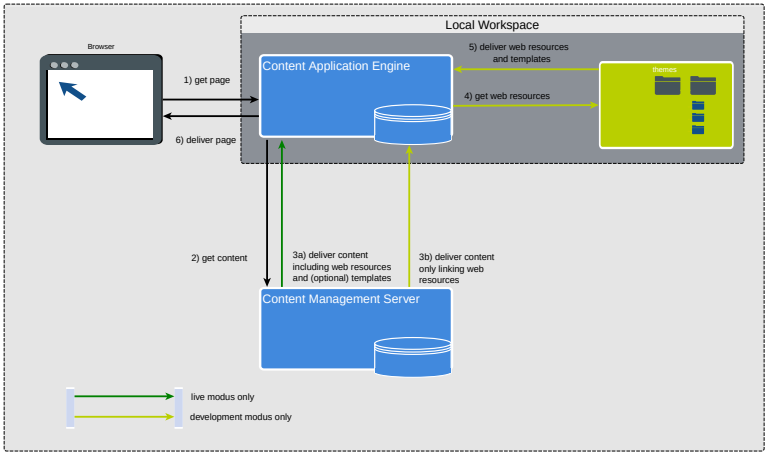

Figure 3.3. Content Application Engine flow in detail

Figure 3.3, “Content Application Engine flow in detail” [25] gives an overview of the idea behind local resources.

1. The browser requests a page from the locally started Content Application Engine.
2. The *CAE* requests the content from the Content Server.
3. While in development mode, the Content Server delivers content such as Articles and content items which link to the web resources.

   In live mode, the Content Server also delivers the web resources to the *CAE*.
4. The *CAE* has got the editorial content which links to the web resources. Now, the *CAE* resolves the local location of the web resources and requests the resources from the file system.
5. The *CAE* reads the resources from the file system.
6. The *CAE* combines the content from the Content Server and the web resources from the file system and delivers the requested page to the browser.

Preparing the Preview

Immediate preview of your changes requires a local preview *CAE* in development mode and the usage of local resources.

Internally, the *CAE* handlers and link schemes will map the linked resource objects of a page content in the repository to the files in the local workspace. For this, you have to do the following configuration:

1. If you're using the Frontend Workspace, all paths are preconfigured and work out of the box.
2. To start the local Spring Boot application in development mode add your CMS host to the `private` Spring profile and use the following command in module `cae-preview-app`

   ```
   mvn spring-boot:run -Dspring-boot.run.profiles=dev,local,private
   ```

   a. The Spring property `delivery.local-resources` of the preview *CAE* must be `true` in order to use local resources. This is the default setting.

   b. The Spring property `delivery.developer-mode` of the preview *CAE* must be `true` in order to run in developer mode. This is the default setting.

   Open your browser at **http://localhost:40980/blueprint/servlet/<YourDemoSite>**.
3. You have to create and link a content structure in the Content Server which corresponds to your local resource structure. The easiest way is to import your resources in the content repository as described in Section 5.6, "Importing Themes into the Repository" [73] and link them afterwards to the site.
4. In order to see the effect of your changes, you have to build your resources after each change. The easiest way is to use `pnpm start --local`. This will watch your Sass, JavaScript and FreeMarker source files and will recompile them after each change.

   Change the developer workflow to "local" in `config/env.json`, since "remote" is the default, if you want to use `pnpm start` instead of `pnpm start --local`.

   ```
   {
     "monitor": {
       "target": "local"
   ```

```
  }
}
```

Editing Source Files

Preview Changes

In general, editing a theme is a straightforward development task as soon as you have set up the preview. When you edit CSS files, Sass files or JavaScript files, add images and, maybe, write FreeMarker templates you will immediately see all changes in your preview *CAE*.

All CoreMedia themes provide a `start` script, which starts the monitor mode. This also includes live reloading to automatically reloading your changed files.

However, before you can start editing a theme, you need a theme. You can either edit an existing theme, or create a new theme. Creating a new theme requires additional work, because before you can see the preview, you need to create a new module, do an initial upload of your theme to the *Content Server* and link it to a site.

NOTE

Renaming or adding of templates will work smoothly, but deleting a template will not work without clearing the cache. Empty the cache or restart the *CAE* to see the affected changes.

Monitoring Changes

Run `pnpm start` for local development

The monitor mode may be run by executing `pnpm start` from your theme directory. The command watches file changes and updates the theme on the local *CAE*.

A live reload mode to automatically refresh the browser on file changes is included.

NOTE

The file system listeners that automatically rebuild the theme when a file is changed are only active after the initial build has finished. This means that if you change any files during the initial build it will not cause the changes to be detected. Better wait for the console output stating "Webpack is watching the files..." before performing any further changes after starting the monitor mode.

The monitor mode may be configured in an `env.json` file in the config directory of the Frontend Workspace using the options listed below. All CoreMedia Themes are pre-configured and work out of the box.

Option	Type	Default	Description
`target`	String	remote	Set this option to `local` in order to configure the monitor mode for a local preview Content Application Engine.
`livereload.host`	String	localhost	This defines the host of the live reload server.
`livereload.port`	Number	35729	This defines the port the live reload server listens on.

Table 3.4. Options to configure the monitor mode

Example

```
{
  "monitor": {
    "target": "local",
    "livereload": {
      "host": "localhost",
      "port": 35729
    }
  }
}
```

To quit a running monitor mode, press the keys`<Ctrl>+<C>`

Studio Preview

When you have configured the preview, you will see the effect of changed web resources in the Content Application Engine in your local browser by navigating through the site that you have changed.

Preview in local Studio

When you have started a local CoreMedia Studio you can watch the changes more comfortably in the Studio preview, because, by default, Studio uses the Content Application Engine for preview which is installed on the same computer as Studio. The Studio preview offers the ability to explicitly search for elements and display them as preview without displaying the surrounding sites while still loading dependencies like CSS styles from web resources.

Preview without local Studio

When you do not want to build and start a local Studio, you can just copy and paste the preview URL of a non-local Studio to a new browser window/tab and change the hostname to your localhost. Therefore, you will see the preview as it would be in *Studio*.

4. Workspace Concept

This guide explains concepts, structure and the functionality of the Frontend Workspace and its provided packages.

4.1 Structure of the Workspace

Root

The workspace root is a package which provides several command line scripts to create a new theme, build themes and execute tests. It has the following file structure:

```
frontend/
  ├── bricks/               // own bricks and example bricks
  ├── config/               // configuration for the development workflow
  ├── lib/                  // API bricks and tools
  ├── node_modules/         // dependencies managed by the package
  │                            manager generated during installation
  ├── src/                  // files for code completion in IntelliJ IDEA
  ├── target/               // target folder for the bundled theme
  ├── themes/               // themes containing CSS, JavaScript,
  │                            templates and other static files
  ├── .eslintrc.json        // eslint configuration
  ├── .gitignore            // specifies files to ignore by git
  ├── package.json          // meta data about the workspace for the
  │                            package manager
  ├── pnpm-lock.yaml        // pnpm lock file to fixate versions
  ├── pnpm-workspace.yaml   // pnpm workspace configuration
  ├── pom.xml               // meta data about the workspace for
  │                            code completion in IntelliJ IDEA
  └── README.md
```

Example 4.1. File structure of the workspace

Please note, that the `config` folder will only be created after running **pnpm start** or **pnpm run deploy** in the Frontend Workspace for the first time.

Available Scripts

You may use the following commands:

Command	Description
`pnpm install`	Downloads and installs all dependencies defined in the `package.json`.
`pnpm test`	Executes `test` scripts which may be defined in `package.json` of each theme and brick in the `themes` or `bricks` directory.
`pnpm build`	Executes the `build` script of all theme packages found directly below `themes/`.

Command	Description
`pnpm run deploy`	Executes the `build` script of all theme packages found directly below `themes/` and uploads it to the `/Themes` folder in the content repository.
`pnpm run create-brick <name>`	Executes the `create-brick` script to generate a new Hello-World brick. See Section 5.2, "Creating a New Brick" [64].
`pnpm run create-theme <name>`	Executes the `create-theme` script to generate a new blank theme. See Section 5.1, "Creating a New Theme" [62].
`pnpm run eject`	Executes the `eject` script to eject an example brick. See Section 5.4, "Using an Example Brick" [69].

Table 4.1. Available Commands

NOTE

You can run `pnpm run` to get a list of all available run-scripts.

Packages

Several other packages can be found in `lib`, `bricks` and `themes` which can be split into four different groups:

Group	Location	Description
API Bricks	lib/bricks	These packages are meant to be used in your themes and bricks to activate different features. They contain various assets (JavaScript, SCSS, Templates, ...) and provide mostly core functionality. See Section 4.3, "Bricks Structure" [39] and Section 6.3, "Bricks" [121].
	bricks	Custom bricks should only be created in the `/bricks` folder. See Section 6.4, "Example Bricks" [140] Section 5.2, "Creating a New Brick" [64] to learn more about creating new bricks. It also contains the example bricks, which are not meant to be used directly in your theme, since they can be changed or removed in new releases without warning. Rather than providing a large set of configuration via parameters, variables and settings they are meant to be changed directly by creating a copy (see Section 5.4, "Using an Example Brick" [69]).

Group	Location	Description
Tools	lib/tools	These packages provide modules and scripts to analyze, customize and build the workspace.
Themes	themes	A theme is meant to compose various bricks, its own assets and customizations as well as other third-party integrations into a bundle by using the tools which can be then be used by the *CoreMedia CAE* to render sites and their underlying content. The existing themes are examples for different integrations. See Section 4.2, "Theme Structure" [35] and Section 6.1, "Example Themes" [103].

Table 4.2. Groups of packages

CAUTION

Do not change or modify any of the files in the provided packages. While API bricks are meant to be used as they are, themes and example bricks should either be copied and customized or you can create your own blank theme using the theme creator. See Section 5.1, "Creating a New Theme" [62]. Otherwise, it can be very hard to upgrade the frontend workspace!

The type of package has to be defined in the `package.json` entry `type` inside `coremedia` and is used by the package `@coremedia/tool-utils`. The following types exist:

Type	Description
`workspace`	Should be set in the root `package.json` to define the workspace. Do not forget to define the workspaces for pnpm too.
`brick`	This type is mandatory for bricks. It is used by the tools to calculate the dependencies.
`lib`	Use for libraries, which are not bricks or themes. It is used by the tools to calculate the imports.
`theme`	This type is mandatory for themes. It is used by the tools to bundle a theme.

Table 4.3. Types of CoreMedia specific packages

In addition to the `type` entry the following entries exist:

Entry	Description
`init`	Indicates the initialization script for the CoreMedia package which is automatically imported when loading the brick (smart import). (Optionally)
`smartImport`	Indicates in which contexts the smart import mechanism will apply, if not set the `"default"` variant will be used meaning it will be applied whenever the theme is loaded. (Optionally)
`shim`	Indicates a mapping for modules to be shimmed. (Optionally) See Section 5.13, "Integrating Non-Modular JavaScript" [83] for more details.

Table 4.4. Entries of CoreMedia specific packages

```
"coremedia": {
  "type": "brick",
  "init": "src/js/init.js",
  "smartImport": [
    "default",
    "preview"
  ]
}
```

Example 4.2. Example configuration of `@coremedia/brick-utils`

The following diagram demonstrates the intended relations between the different package groups including external packages:

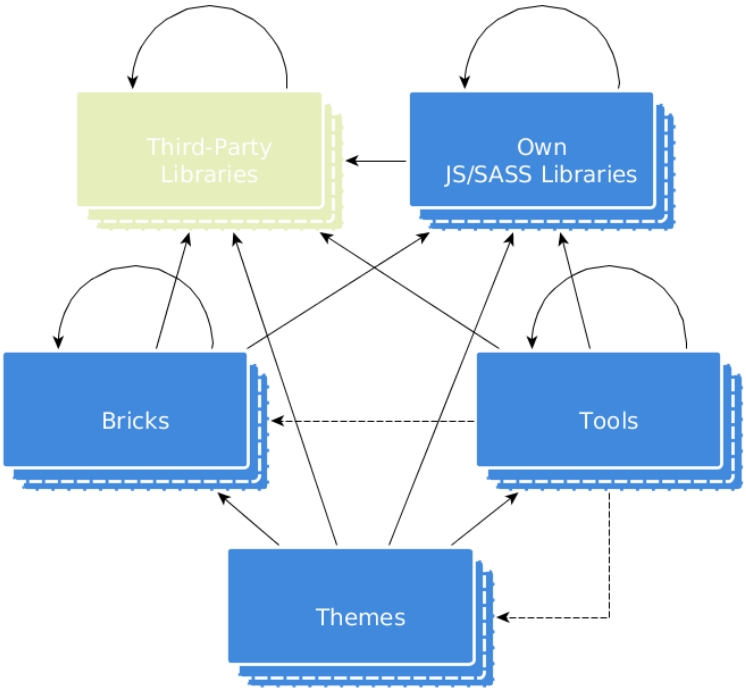

Figure 4.1. Relations between package groups.

`Bricks` may also include external third-party libraries if necessary (for example, jQuery or bootstrap-sass). A `brick` never depends on a `theme` or the `tools` but may be based on another `brick` where it makes sense.

While packages of the `Tools` group know about the general structure of `bricks` and `themes`, they will never directly depend on a concrete `brick` or `theme` package (though only indicated by a dotted arrow).

`Themes` may depend on everything else in the workspace as well as external third-party libraries, but they should never depend on each other as they are meant to be the endpoint of the hierarchy where the build process is triggered. An exception are child themes that are derived from another theme. For more information see Section 5.5, "Theme Inheritance" [71].

4.2 Theme Structure

A theme is a package which consists of a theme config, a webpack configuration and various web resources located in its `src` folder. Example 4.3, " Filesystem structure of a theme " [35] shows the filesystem structure of a theme:

```
themes/
└── [$example-theme]/      // name of theme,
    │                         for example "foo" and suffix "-theme"
    ├── node_modules/      // installed node dependencies managed
    │                         by the package manager
    ├── src/               // all source files, add subfolders for all types
    │   │                     like sass, js, fonts or images
    │   ├── fonts/
    │   │   └── example.woff
    │   ├── img/
    │   │   ├── example.png
    │   │   └── logo.svg
    │   ├── js/
    │   │   ├── index.js
    │   │   ├── [$example-theme].js
    │   │   └── preview.js
    │   ├── l10n/
    │   │   ├── [$example-theme]_de.properties
    │   │   └── [$example-theme]_en.properties
    │   ├── sass/
    │   │   ├── partials/
    │   │   ├── variables/
    │   │   ├── [$example-theme].scss
    │   │   ├── _partials.scss
    │   │   ├── _variables.scss
    │   │   └── preview.scss
    │   ├── settings/
    │   │   ├── [$example-theme].settings.json
    │   │   └── Preview.settings.json
    │   └── templates/
    ├── .prettierignore    // path configuration for prettier
    ├── .prettierrc        // config for prettier
    ├── package.json       // config for node module dependencies
    ├── theme.config.json  // theme definition config, mandatory
    │                         for importing themes!
    └── webpack.config.js  // config for webpack
```

Example 4.3. Filesystem structure of a theme

Theme config

The theme config is a JSON file named `theme.config.json` located in the root directory of the theme package. It contains general information like the name and a description of the theme but also path references to all its web resources (JavaScript, CSS files, Templates, ...). Example 4.4, " Theme config example " [35] shows the typical structure of a theme configuration. You can find a reference here: Section 6.2, "Theme Config" [117].

```
{
  "name": "example-theme",
  "description": "The is an minimal example theme.",
  "thumbnail": "src/img/theme-example.png",
```

```
  "scripts": [
    {
      "type": "webpack",
      "src": "src/sass/example.js"
    },
    {
      "type": "copy",
      "src": "src/vendor/example.js"
    },
    {
      "type": "externalLink",
      "src": "https://cdn.example.org/external.js"
    }
  ],
  "styles": [
    {
      "type": "webpack",
      "src": "src/sass/example.scss"
    },
    {
      "type": "copy",
      "src": "src/css/example.css",
      "target": "css/example.css"
    },
    {
      "type": "externalLink",
      "src": "https://cdn.example.org/external.css"
    }
  ],
  "l10n": {
    "bundleNames": [
      "Example"
    ]
  }
}
```

Example 4.4. Theme config example

Webpack configuration

Every theme requires a webpack configuration in order to be build via `pnpm build`. The theme creator (see Section 5.1, “Creating a New Theme” [62]) will create a default configuration in the `webpack.config.js` which makes use of `@coremedia/theme-utils` to apply our default configuration.

Structure of web resources

Before giving more detailed information about the structure of the web resources it is important to note that `Webpack` is used to bundle a theme for deployment. It performs tasks like transpilation (ES6 -> ES2015), compilation (SCSS -> CSS), bundling (ES6 modules -> CommonJS modules) and minification before the theme is uploaded to the CoreMedia repository. Because of this the source file structure of a theme is not kept and one needs to distinct between the theme's `Source File Structure` and its `Bundled File Structure`. More information about the tasks can be found in the corresponding chapters for SASS and JavaScript. For further details about the deployment check Section 5.4.12, “Client Code Delivery” in *Blueprint Developer Manual* .

Source File Structure

All `Source Files`, except for the templates and the theme's main entry points, can be arranged arbitrarily in directories. However, in CoreMedia themes these resources are arranged by their particular types. CoreMedia uses the following typical style for web-safe file names:

- File names should all be lower case

- Nouns should be used in singular
- Words should be separated by dashes

Templates structure

Templates are located in the `src/templates` directory of the theme module. Inside this directory templates are structured in packages, corresponding to the content beans. The order of the elements also specifies the order the JAR files are processed by the CAE. See Section 5.4.10, "Dynamic Templating" in *Blueprint Developer Manual* for details.

Entry Points

The term is based on https://webpack.js.org/concepts/entry-points/ and describes the entry points of the different language layers. CoreMedia themes have two main entry points:

- `src/js/index.js` is the main entry point for JavaScript.
- `src/sass/$theme-name.scss` is the main entry point for SCSS.

Starting from an entry point you can import all other required files.

Bundled File Structure

All templates coming from bricks and themes are bundled into a `templates/$theme-name-templates.jar` archive, while templates from the theme overwrite those from bricks.

The JavaScript will be bundled into `js/$theme-name.js` while the SCSS will be bundled into a CSS file `css/$theme-name.css`.

Web resources will automatically be bundled if they are referenced in the JavaScript or in the SCSS regardless of their location. For convenience all static web resources of a theme under `src/css`, `src/fonts`, `src/img`, `src/images` and `src/vendor` will be copied to the corresponding location in the theme's target folder. However, CoreMedia strongly encourages to always reference the static web resources in one of the entry points as it guarantees that the web resource is bundled properly and the link is properly transformed to the `Bundled File Structure` which may differ from the `Source File Structure`.

Bundled web resources will be bundled by their type:

- `svg`, `png` and `gif` files are placed under `img` keeping their original filename.
- `woff`, `woff2`, `ttf` and `eot` files are placed under `fonts` keeping their original filename.
- `swf` files are placed under `swf` keeping their original filename.

To add more file types to the layout you need to specify an additional `file-loader`.

Themes in the CoreMedia repository

Themes imported into the *Content Server* are stored in a folder named `Themes/<ThemeName>` by default. The content is stored in the following content types:

- CSS files in content of type `CSS`

- JavaScript files in content of type `JavaScript`
- FreeMarker Templates as JAR archives in blob properties in content of type `Template Set`
- Resource bundles in content of type `Resource Bundle`
- All other supported web resources in content of type `Technical Image`

Select the theme as the associated theme for the page content of your site (see Figure 5.2, “Associated Theme” [74]).

4.3 Bricks Structure

Reusability

Bricks are reusable frontend modules for themes. They can contain templates, styles, images, fonts, resource bundles and JavaScript.

Standalone Packages

The idea of bricks is to split frontend features, special views or other functionality, like ImageMaps or Responsive Images into small modules instead of providing a big chunk like a basic theme. Technically, every brick is a package. By declaring everything it requires in its `package.json` (for example, its dependencies to third-party packages or other bricks) a brick is self-contained.

Activation By Dependency

A brick can be used by a theme just by adding it as a dependency in the theme's `package.json`. The build process will provide everything the brick needs in order to be usable.

There are two kinds of bricks in the workspace. API bricks are provided in the `lib/bricks` folder. They are meant to be used directly in your theme or your bricks, and provide core functionality. While some bricks only provide helpers in form of FreeMarker Libraries and SCSS Mixins, some already contain generic views in form of FreeMarker Templates that can be adjusted via template parameters or styling that can be controlled via SCSS variables.

Example bricks are examples of how you can use the Frontend Workspace and API bricks. They mostly contain fully fledged layouts with special behavior in different devices. They are not meant to be used directly in your theme, since they can be changed or removed in new releases without warning. Rather than providing a large set of configuration via parameters, variables and settings they are meant to be changed directly by creating a copy (see Section 5.4, “Using an Example Brick” [69]).

Just like a theme a brick is a package which consists of various web resources located in its `src` folder. It is meant to be a reusable frontend module that is easy to add to a theme without having to know much about its inner structure. Example 4.5, “ File structure of a brick ” [39] shows the filesystem structure of a brick:

```
bricks/
└── [$brick-name]/
    ├── src/
    │   ├── freemarkerLibs/
    │   │   └── [$brick-name].ftl
    │   ├── fonts/
    │   │   └── example.woff2
    │   ├── img/
    │   │   └── example.png
    │   ├── js/
    │   │   └── index.js
    │   ├── l10n/
    │   │   └── [$brick-name]_en.properties
    │   ├── sass/
    │   │   ├── partials/
    │   │   ├── variables/
```

```
    │   │   ├── _partials.scss
    │   │   └── _variables.scss
    │   └── templates/
    ├── .prettierignore
    ├── .prettierrc
    └── package.json
```

Example 4.5. File structure of a brick

Source File Structure

Bricks can provide JavaScript, SCSS, templates, localization and other web resources just like images and fonts. The theme build process knows about the file system layout of bricks so it can easily integrate the different parts into the bundled theme that is used on a website.

Dependency Management

Just like every package bricks can depend on other packages using their `package.json`. As the package.json supports multiple kinds of dependencies CoreMedia encourages using (normal) "dependencies" for most of the use cases (especially when depending on other bricks) and "devDependencies" when requiring specific tools (for example, test frameworks) that should not be installed when just using the brick in a theme or in another brick.

When a brick depends on another brick, it will always include the other brick's web resources, so only direct dependencies need to be handled by a theme.

NOTE

Bricks may not depend on themes but they may depend on other bricks if necessary. If you're creating your own bricks, be aware to avoid cyclic dependencies between them even if this will not break the building of themes. CoreMedia recommends using the script `pnpm create-brick name` to create a new brick, see Section 5.2, “Creating a New Brick” [64].

JavaScript

A brick always provides JavaScript using the "main" entry in the `package.json`. For CoreMedia's bricks `src/js/index.js` is used. In case no "main" entry is provided the lookup mechanism will check if there is a `index.js` directly below the brick folder which is the default behavior of Node JS.

SCSS

Every brick also provides two SCSS files: `_variables.scss` and `_partials.scss` directly below `src/sass/`. The `_variables.scss` represents the variables or configuration layer and only defines variables while never producing any output. The `_partials.scss` represents the partials or output layer which assumes that it is imported after the configuration. It creates the actual CSS rules based on the values of the variables.

The separation of these two layers is crucial and should be taken into account when creating an own brick. More information about the SASS structure can be found in Section 4.4, “Sass Files” [42].

Templates

Just like a theme a brick can provide templates that will be considered by the view lookup mechanism. Templates can be found below `src/templates`. Technically bricks can override the templates of other bricks. The order in which the templates are copied is determined by the dependency tree. Considering a theme is the root, leaf bricks will always be copied first moving the tree down to the root so templates of dependent bricks are always copied before the depending brick.

Localization

Localization follows the same pattern as described in Section 4.6, “Localization” [46]. The resource bundle files can be found directly below `src/l10n/`.

Other web resources

Other web resources just like images and fonts are not just copied into a theme but will be gathered by the theme build process when analyzing the JavaScript and the CSS code produced by the SCSS build. Both types can reference other web resources. While in JavaScript `require` statements are used, in CSS code all data URL directives will be parsed to collect other web resources.

As the location in which the web resources are placed is determined by the build process, bricks do not make any assumptions about the file structure of the bundled theme. This also means that data URLs and require statements are the only place where other web resources are referenced.

CAUTION

To keep the bricks maintainable and easy to upgrade it is highly recommended to make no changes to the files and folders in the `lib/bricks` directory, except creating your own brick. Otherwise, upgrading via a patch file may no longer be possible.

4.4 Sass Files

In the *CoreMedia Blueprint* themes CSS files are generated from Sass files (see sass-lang.com). Except for the root Sass files of a theme (`$theme-name.scss` and `preview.scss`) which are also called `entry points` all files should start with an underscore which tells the Sass compiler that the generated code will not be written into a separate CSS file but into the same output file as the Sass file it was included from. Sass Files starting with an underscore that generate styles when importing them are called `Partials`.

The folder structure is as follows:

```
sass/                   // sass files are located inside the themes 'src'
folder
├── partials/           // contains partials
│   ├── _grids.scss
│   ├── _banner.scss
│   └── ...
├── variables/          // contains configuration in form of variables
│   ├── _colors.scss
│   ├── _grids.scss
│   ├── _variables.scss
│   └── ...
├── [$theme-name].scss  // main entry file for a theme
└── preview.scss        // preview entry file for a theme (will be covered
later)
```

Example 4.6. Folder structure of the Sass files

All imports of variables need to be performed before any partial is being imported in the main entry file. This leads to the following import order (based on the folder structure in the last section):

```
// ### VARIABLES ###

@import "variables/bootstrap_variables";
@import "variables/grids";

// ### PARTIALS ###

@import "partials/grids";
@import "partials/hero";
```

Example 4.7. Import order in entry files of a theme

Importing SASS Code of Bricks

Sass code in `Bricks` follows the same patterns as Sass code in themes, so the code is also split up into `variables` and `partials`. When importing a brick, you also need to make sure that all variables are loaded before any partials. For this every brick

provides a `_variables.scss` and a `_partials.scss` that are meant to be imported keeping the order in mind. A typical import of bricks using the "smart-import" mechanism looks as follows:

```
// ### VARIABLES ###

// Own variables

@import "variables/bootstrap_variables";
@import "variables/grids";

// dependency variables

@import "?smart-import-variables";

// ### PARTIALS ###

// dependency partials

@import "?smart-import-partials";

// Own partials

@import "partials/grids";
@import "partials/hero";
```

Example 4.8. Import order in entry files of a theme with bricks

Your own variables need to be set before any of the brick variables will be included. The reason behind this is that in sass its common practice to use the `!default` flag (see: Variable Defaults: !default). for variables that are meant to be configurable. As bricks in most cases provide configuration in their Sass code, you need to override configuration before their variables are imported.

The file names `?smart-import-variables` and `?smart-import-partials` are only placeholders. They do not actually exist but will be substituted by our theme build mechanism to include all dependencies that work with the "smart-import" mechanism. Please note that - as CoreMedia cannot make any assumptions about the structure of third-party libraries - the "smart-import" mechanism will only work with modules that have an entry `coremedia.type` in their `package.json` that is either set to `lib` or `brick`.

CAUTION

While also being convenient these two imports serve as a contract between a brick and a theme. A brick always expects that its Sass files are imported. This means that whenever you add a brick to your theme (by adding a dependency to its package.json) you need to import its `_variables.scss` and `_partials.scss` using the above code in your main entry sass file.

You do not need to take care of the dependencies a brick might bring in turn (transitive dependencies). This is also handled by the brick as it will already import the variables and partials of its dependencies. You also do not need to worry that code might be imported twice, only the first import of a Sass file will by performed, all later imports of the same files are ignored.

Suggestions for Preview Specific Styles

Currently, there is no deep integration for preview specific adjustments to a theme - this includes preview specific styles. The only thing that the default theme build provides is the possibility to place a `preview.scss` next to the `$theme-name.scss` that will be compiled into a `preview.css` in the theme's target folder next to the `$theme-name.css`.

The preview entry file should follow the same patterns as the main entry file but should be treated as an addition rather than a complete separated entry. While the main entry file (`$theme-name.scss`) should import all variables and partials required for the theme (including dependencies) the preview entry file (`preview.scss`) may import common variables that are also used in a theme but should never include partials (and so code generating files) that are already part of the theme. As CoreMedia only supports preview specific as an addition to the existing styles of the theme this may lead to code duplication or even unintended overrides (if the variables configuration is different, for instance).

The "smart import" mechanism also works for the `preview.scss` and will handle the import of preview specific bricks (which currently leads to importing the SCSS code from `@coremedia/brick-preview`).

4.5 Images

There are no special rules for images. Images are imported in `Technical Image` content items. In your CSS or JavaScript files in the workspace, you link to images through a relative path URL. For example, `background-image: url("../images/testimage.png")`. After the upload, these links are replaced by internal content links.

The following image types in themes are supported: jpeg, gif, png, svg, webp, and avif.

NOTE

Inside themes images can also be referenced from FreeMarker templates (see Section 5.7, “Referencing a Static Theme Resource in FreeMarker” [76]).

4.6 Localization

Resource bundles

Sometimes a template needs to render localized text that is not part of the content (for example when rendering descriptive information). As templates are meant to be independent of a specific language, a mechanism has been added to render localized texts by using a unique, symbolic name instead of the actual text in templates.

To be able to achieve this, every brick and theme package can provide these unique keys in form of one or more resource bundles placed in the `src/l10n` folder of the package. Resource bundles are Java Properties files that follow a certain naming pattern, for example:

- `my-theme_en.properties`
- `my-theme_de.properties`

This means according to the name of the file that you have a set of resource bundles named `my-theme` which provides localization for two different languages: "en" and "de" (represented as ISO 639-1 code by the suffix of the basename, see ISO 639-2 Language Code List). A set of resource bundles always needs a `master resource bundle` which is used as a fallback if no other localization is found. For our theme's and bricks this is the English localization. The resource bundles for other languages than the master are called the `derived resource bundles`.

To add a resource bundle to a theme, it has to be added to the theme configuration. See Section 6.2, “Theme Config” [117]:

```
{
  "name": "my-theme",
  ...
  "l10n": {
    "bundleNames": [
      "MyTheme"
    ]
  }
  ...
}
```

All properties files contain pairs of keys and values where the key is the symbolic name used in the template and the value is the text localized for a concrete language. The identifiers used as a key are restricted to certain letters (for example, no spaces can be used). For more information about the syntax check Properties File Format. By default, the ThemeImporter assumes the properties files to be Latin-1 encoded. If you store them in a different encoding (like UTF-8), you must specify the encoding in the theme configuration. For details see Section 6.2, “Theme Config” [117]. The master (in this case "en") properties file might look like this:

```
# this is the english properties file
button_close=Close
```

```
info=Info
search_results=There are {0} search results for the term "{1}"
```

While the derived "de" properties files might look like this:

```
# this is the german properties file
close=Schlie\u00DFen
search_results=Es gibt {0} Such-Ergebnisse für den Suchbegriff "{1}".
```

As you can see, the derived properties file does not contain all keys the master file has. This is okay as the lookup mechanism will always fall back to the master properties file in case the key was not found in the resource bundle of the concrete language.

NOTE

While it is okay to omit keys of a master resource bundle in a derived resource bundle, this does apply to the other way around. A derived resource bundle should never define a key the master resource bundle does not provide.

Resource bundles of bricks are aggregated and merged based on the dependencies added to the `package.json` of the theme. When including any brick just add the `Bricks` resource bundle name to your theme configuration and the localization of all bricks is added. For more information see Section 5.3, "Using Bricks" [67].

Key names are unique across sets of resource bundles of all packages. Avoid using the same key in different packages as long as you are not overriding a key assigned. In case the same key is used in multiple different sets of resource bundles the order in which the resource bundles are added in the theme config is important as the first assignment of the key determines the value. All following assignments are ignored. This also applies if (for whatever reasons) a key is defined multiple times in the same file but it will also log an error when importing the theme into the content repository. As the resource bundles of bricks are merged into a single resource bundle make sure that you use unique keys across all bricks, overriding existing keys in bricks is not supported.

Usage in templates

After a message key is defined for different languages, it can be used in the template in two different ways using the FreeMarker facade described in Section 6.5.1, "CoreMedia (cm)" [175].

- `<@cm.message key args escaping />`
- `${cm.getMessage(key, args)`

Both methods are wrappers for `springMacroRequestContext.getMessage()` of the Spring Framework and support optional arguments. Please also take a look at the official spring.ftl descriptions.

NOTE

When not using the Chapter 3, *Web Development Workflow* [19], make sure that you upload the theme to the content server when adding a new resource bundle to the theme config before using it in the template. Otherwise, the resource bundle will not be taken into account when accessing a key in the template.

4.7 Settings

Settings

Some settings can be clearly assigned to a specific theme or brick. Some of these settings might even only make sense in the context of a specific theme or if certain bricks are active. These settings would probably need to be changed if a different theme is chosen, for example, for a sub page. Because of this, settings can now also be declared within the frontend workspace.

To be able to achieve this, every brick and theme package can provide one or multiple settings files placed in the `src/settings` folder of the package. Settings are `JSON` files which end with `.settings.json`:

- `MyTheme.settings.json`
- `Preview.settings.json`

A typical settings file looks like this:

```
{
  "sliderMetaData": {
    "cm_responsiveDevices": {
      "mobile": {
        "width": 414,
        "height": 736,
        "order": 1
      },
      "tablet": {
        "width": 768,
        "height": 1024,
        "order": 2
      }
    },
    "cm_preferredWidth": 1280
  },
  "fragmentPreview": {
    "CMPicture": [
      {
        "titleKey": "preview_label_teaser",
        "viewName": "asTeaser"
      }
    ],
    "CMTeasable": [
      {
        "titleKey": "preview_label_default",
        "viewName": "DEFAULT"
      },
      {
        "titleKey": "preview_label_teaser",
        "viewName": "asTeaser"
      }
    ]
  }
}
```

Example 4.9. Preview.settings.json

Supported Property Types for Settings

A settings file will be imported to the content server when a theme is deployed and is stored in a `CMSettings` content item linked to the theme. As the content type uses `Structs` settings files can declare the following types:

```
{
  "my-string-property": "Hello World",
  "my-string-list-property": ["Hello", "World"]
}
```

Example 4.10. String / String List

```
{
  "my-integer-property": 1,
  "my-integer-list-property": [0, 9]
}
```

Example 4.11. Integer / Integer List

```
{
  "my-boolean-property": true,
  "my-boolean-list-property": [true, false, false]
}
```

Example 4.12. Boolean / Boolean List

```
{
  "my-link-property": {
    "$Link": "../sass/styling.scss"
  },
  "my-link-list-property": [
    {
      "$Link": "../sass/styling.scss"
    },
    {
      "$Link": "../sass/more-styling.scss"
    },
  ]
}
```

Example 4.13. Link / Link List

```
{
  "my-date-property": "2018-11-13",
  "my-date-list-property": [
    "2018-11-13 20:20:39",
    "2018-11-13+03:00",
    "2018-11-13 20:20:39-09:00"
  ]
}
```

Example 4.14. Date / Date List

```
{
  "my-struct-property": {
    "hello": "world",
    "show": true
  },
  "my-struct-list-property": [
    {
      "nestedStruct": {
        "hello": "world"
      }
    },
    {
      "list": [1, 2, 3]
    }
  ]
}
```

Example 4.15. Struct / Struct List

As you can see it is basically plain `JSON` syntax except for link and date properties (and their list counterparts). You can describe almost everything that can be expressed via `JSON` with settings files. However, there are the following limitations:

- Property names may not contain a colon [":"].
- Lists cannot have mixed item types. This is because structs are also restricted to this, otherwise the settings could not be imported to the content server.
- List may not be empty as otherwise the list type that needs to be declared in the struct cannot be detected.
- Links can only point to scripts and styles that are defined in the theme configuration.

If one of the limitations is neglected the theme build will trigger a warning or an error accordingly.

Merging of Settings

During the theme build the settings files of all packages will be aggregated and merged. Merging is performed on filename base, so all settings files of the same name in different packages are merged into a single settings file with that name. If a setting is declared multiple times, the setting that is declared closer to the root of the theme's dependency tree takes precedence. This is the same mechanism as for SCSS variables and templates.

Properties are always overridden except for struct properties. If a struct property is encountered multiple times, the theme build will merge the structs instead of replacing the former ones entirely. This is a deep merge, so nested structs will also be merged.

CAUTION

While you can have multiple settings files to structure the settings to your needs you need to make sure that if the same top-level property is used multiple times in different packages it is always declared in the same settings file.

Let's assume the example for `Preview.settings.json` at the beginning of the chapter is declared in the preview brick. In case you want to override the `fragmentPreview` and `sliderMetaData` in a brick or theme that depends on the preview brick you need to create a `Preview.settings.json` file in the `src/settings` folder of your theme.

Settings Lookup

Settings can be looked up in FreeMarker Templates using the `bp.setting` method of the Section 6.5.3, "Blueprint (bp)" [185]. The lookup mechanism for the given key will first check the given `self`, then the `context` (for example, the cmpage) and finally the theme. This implies that a theme setting has the least precedence of all settings definitions and will only be taken into account if it is not overridden somewhere along the lookup chain.

If you want to use theme settings in other backend modules (for example, content beans) via the `com.coremedia.blueprint.base.settings.SettingsService` you need to make sure that the theme is actually taken into account when providing beans for the lookup. Please check the `com.coremedia.blueprint.cae.web.taglib.BlueprintFreemarkerFacade` to find code examples on how to achieve this.

4.8 Templates

Dynamic templating (see Section 5.4.10, "Dynamic Templating" in *Blueprint Developer Manual*) requires the usage of FreeMarker, not JSP, templates. FreeMarker templates are imported as JAR files into a blob property of content of type `Template Set`. See Content Application Developer Manual for more details about templates.

Templates Naming and Lookup

The view dispatcher of the CAE (see the Content Application Developer Manual for more details) selects the appropriate view template for a content bean according to the following data:

1. Name of the content bean

   The view dispatcher looks for a template whose name starts with the name of the content bean.

   Example: The template `CMExample.ftl` is a detail view for the content bean `CMExample`.

2. A specific view name

   A view name specifies a special view for a content bean. The view is added as a parameter when you include a template in another template via `<cm.include self=self view="asContainer"/>`.

   Example: The template `CMExample.specialView.ftl` is a special view for the content bean `CMExample`.

3. A specific view variant

   A view variant is used, when the look of a rendered view should be editable in the content (see Section 5.4.7, "View Types" in *Blueprint Developer Manual* for details).

   Example: The template `CMExample.[differentLayout].ftl` is a special view of the content bean `CMExample`. The view variant must be enclosed in square brackets.

The template name is always in the order content bean name, view name, view variant. The view dispatcher looks for the most specific template.

FreeMarker

Escaping HTML Output

In *CoreMedia Blueprint* escaping of templates is enabled to prevent output that allows cross-site scripting (XSS) attacks. The default output format for all templates is set to HTML. See FreeMarker online documentation for details.

In special cases, it might be necessary to disable escaping. For this purpose, FreeMarker provides the directive `<#noautoesc/>` or built-in for Strings `?no_esc`.

> **CAUTION**
>
> Note that disabling HTML escaping can lead to cross-site scripting (XSS) vulnerabilities if a templates outputs unchecked data like user input that may contain scripts.

Robustness of Templates

In order to make sure that the rendering of templates does not fail you have to ensure that FTL templates can be rendered, although some information is not provided. In order to achieve this, FreeMarker adds some functionality to detect if a variable is set and if it contains content.

If you want to check for existence and emptiness of a hash/variable (null is also considered as empty) you need to use `?has_content`.

If you want to declare a default value for an attribute that could be null or empty use `!` followed by the value to be taken if the variable/hash is null.

Example:

```
${existingPossibleNullVariable!"Does not exist"}
<#list existingPossibleNullList![] as item>...</#list>
```

Example 4.16. Example of a fallback in FreeMarker

FreeMarker for JSP Developers

As a JSP developer you are familiar with JSPs in general and with writing CAE templates with JSPs. In this section, you will learn about important differences.

Type-Hinting

Type-hinting in JSP or FreeMarker templates helps IntelliJ Idea to offer you code completion and to make the templates "green". The syntax of the required comments differs between FreeMarker and JSP:

- Comments are marked with `<#-- comment -->` instead of `<%-- comment --%>`
- The annotation is called `@ftlvariable` instead of `@elvariable`
- The attribute that names the typ-hinted object is called `name` instead of `id`
- The comment must have a single space after the opening comment tag

```
JSP:
<%--@elvariable id="self" type="com.coremedia.blueprint.MyClass"--%>

FreeMarker:
<#-- @ftlvariable name="self" type="com.coremedia.blueprint.MyClass" -->
```

Example 4.17. Difference between JSP and FreeMarker type-hinting comment

> **CAUTION**
>
> Code completion only works out-of-the-box when using the *CoreMedia Blueprint* workspace. In addition to this you need to enable the Maven profile `code-completion` in your IDE.

Passing Parameters

In JSP files, it was necessary to wrap arguments passed to taglibs or other functionality into quotes and to print them out via `${}`. In FreeMarker, this is no longer necessary.

```
JSP:
<mytaglib:functionality name="${name}" booleansetting=true />

FreeMarker:
<mymacro.functionality name=name booleansetting=true />
```

Example 4.18. Passing parameters

4.9 Sharing FreeMarker Functionality

FreeMarker templates can be shared among other packages in the Frontend Workspace to reuse functionality. In most cases you may want to utilize FreeMarker functions or macros to define a functionality so that it can be imported by another template using the import directive. These templates will be referred to with the term FreeMarker Library.

Location of FreeMarker Libraries

Shared templates should be located in `src/freemarkerLibs` of your package to be handled specially by our theme build process. They may have any valid file name but CoreMedia suggests naming them after the functionality or package it is provided by using dash-case to separate words.

> NOTE
>
> Some of the CoreMedia packages provided in the frontend workspace already contain FreeMarker libraries. They are documented in the Section 6.5, “CoreMedia FreeMarker Facade API” [175].

Importing a FreeMarker Library

Importing a FreeMarker library in the same package it is provided by is straight forward. Assuming you have a FreeMarker library named `src/freemarkerLibs/my-lib.ftl` which provides a function named "calculateSomething" and a macro called "renderSomething" it can be imported from another template within the same package using the following code:

```
<#import "../../freemarkerLibs/my-lib.ftl" as myLib />

<#assign result=myLib.calculateSomething() />
<@myLib.renderSomething />
```

Example 4.19. Import from src/templates/com.coremedia.blueprint.common.contentbeans/CMArticle.ftl using relative path

In case you want to reference a FreeMarker Library from another package you first need to add a dependency to the other package in its `package.json`. Assuming the

FreeMarker library of the previous example is in a package named "my-freemarker-lib" the template can then be imported with the following code:

```
<#import "*/node_modules/other-package/src/freemarkerLibs/my-lib.ftl" as
myLib />

<#assign result=myLib.calculateSomething() />
<@myLib.renderSomething />
```

Example 4.20. Import from any other template using acquisition

NOTE

The acquisition feature of FreeMarker's `include` and `import` directives are used here to achieve the same lookup mechanism that *Node.js* uses. When building a theme these paths are automatically rewritten so they represent the actual location in the JAR file that is uploaded into the blog property of the `Template Set` (see Section 4.8, “Templates” [53]).

4.10 Upgrading the Workspace

A convenient way to update your frontend workspace is by using a Git patch file, generated from the Coremedia Frontend Workspace for Blueprints GitHub repository.

Generating the patch file

CoreMedia recommends creating the patch file via GitHub. The releases for various AMP and AEP are listed in our frontend repository and their tags can be used in the URL scheme below. Upon entering this in your browser the patch file will be generated immediately.

```
https://github.com/coremedia-contributions/coremedia-frontend-workspace-for-blueprints/compare/
  \
<version to upgrade from>...<version to upgrade to>.patch
```

For example `/cms-9-1801.2...cms-9-1801.4.patch`

Applying the patch

To apply the patch to your workspace place the patch file in its root directory and use your IDE or the following Git command:

```
git apply <filename>.patch
```

This will include the patch as unstaged changes in your current branch. To apply the patch as a commit, please use `git am`. To only list the changes, add the `--check` option. For more information please visit the Git Documentation.

When you successfully upgraded the workspace make sure to follow the release and upgrade notes for every version the patch contains.

CAUTION

In order to minimize conflicting changes when applying the patch file, files and folders of the frontend workspace inside the `lib` folder should remain untouched. For more information on how to add your own bricks or themes have a look at Section 5.3, “Using Bricks” [67] or Section 5.1, “Creating a New Theme” [62].

If you removed the themes provided by CoreMedia from your workspace, applying the patch can run into errors. A workaround is to use the `--exclude=[path]` option and exclude the themes folder. Otherwise, the task can fail.

4.11 Browser Support

CoreMedia supports and tests the bricks and themes provided by the Frontend Workspace for the latest version of the following browsers:

- Chrome
- Firefox
- Edge

For more information about the environments CoreMedia supports, please check the Supported Environments PDF from the documentation.

Browserslist Settings

When bundling a theme, the `browserslist` setting of its `package.json` is taken into account. All example themes (see: Section 6.1, "Example Themes" [103]) have set the browserslist according to our supported environments:

```
"browserslist": [
"last 1 Chrome version",
"last 1 Firefox version",
"last 1 Edge version"
]
```

In the build process of the theme the browserslist is taken into account for bundled `CSS` and `JavaScript` using `Webpack` loaders. This will affect the generated output so the corresponding asset can be parsed by browsers that did not (fully) support certain language constructs.

`CSS` is transformed using the `postcss-loader` in the loader chain for SCSS files (see Chapter 4, *Workspace Concept* [29]). The `autoprefixer` plugin is used that takes the browserslist configuration into account. This means that you don't need to add any browser specific prefixes to your SCSS code and it will also remove browser specific prefixes that are not needed (for example, when embedding third-party code that you probably do not want to customize to add or remove prefixes).

The transformation of `JavaScript` is similar. In this case the `babel-loader` is used in the loader chain for JS files (see Chapter 4, *Workspace Concept* [29]) which supports a `browserslist` configuration via a *Babel* preset called `babel-preset-env`. This means you can write JavaScript in your theme using new ECMAScript syntax and to a certain extent also features and the code is transpiled down to a proper language level when the theme is build so every browser matching the configuration is supported.

You can adjust the settings to your needs. If no setting is provided, it will fall back to the `browserslist` default. For more information about browserslist and its configuration see: github.com/ai/browserslist.

CAUTION

Changing the `browserslist` configuration does not mean that the theme now out-of-the-box supports all browsers that match the given expressions. It only makes sure that the node modules affected by this configuration (see above) will transform the corresponding asset to a common language level that all browsers support.

`autoprefixer` will not add any feature support to browsers. For example if you want to enable support for `flexbox` you will need to add a proper JavaScript polyfill.

`babel-preset-env` will also add some polyfills to add browsers support for a specific feature but this will not cover every feature (for example, a polyfill for `Promises` is not added in the currently used version). You still need to test the theme in the added browsers and probably need to add polyfills accordingly for features the transpiler does not handle out of the box.

5. How-Tos

This section describes how to handle common use cases when working with the Frontend Workspace. It provides some examples and links to the in-depth chapters for further information.

5.1 Creating a New Theme

The CoreMedia Frontend Workspace provides a script to easily create a new minimum theme skeleton, including brick configuration and theme inheritance. It works on macOS, Windows, and Linux.

Quick Overview

```
pnpm install
pnpm run create-theme <name>
pnpm install
cd themes/<name>-theme
```

Installation

After running `pnpm install` the script is ready to be used like all provided scripts.

You'll need to have Node = 20.x on your machine. You can use nvm to easily switch Node versions between different projects.

This tool does not need a Node backend. The Node installation is only required for tooling.

Usage

To create a new theme, run (replace <name> with a name according to the rules below):

```
pnpm run create-theme <name>
pnpm install
cd themes/<name>-theme
```

It will create a directory with the pattern `<name>-theme` inside the themes folder, after asking for some configuration.

The tool lets you decide which bricks you want to include into your dependencies when creating the theme and asks if you want to keep the unused bricks as commented out dependencies in your newly created theme. It also allows you to select an existing theme as the parent of the new one. Learn more about how to extend themes in Section 5.5, "Theme Inheritance" [71].

Inside that directory, it will generate the initial theme structure as described in Section 4.2, "Theme Structure" [35].

You´ll need to run `pnpm install` from the root of the frontend workspace to install the dependencies of the new theme before the theme can be used.

NOTE

The theme name should be a simple ASCII name. Whitespace and special characters are stripped and the name will be lowercase.

5.2 Creating a New Brick

The CoreMedia Frontend Workspace includes a script to easily create a new Hello-World brick. It comes with everything needed to work with templates, JavaScript files, style sheets and localizations. The script works on macOS, Windows, and Linux.

Quick Overview

```
pnpm install
pnpm run create-brick <name>
pnpm install
cd bricks/<name>
```

Installation

After running `pnpm install` the script is ready to be used like all provided scripts.

Usage

To create a new brick, run (replace <name> with a name according to the rules below):

```
pnpm create-brick <name>
pnpm install
cd bricks/<name>
```

> **NOTE**
>
> The brick name should be a simple ASCII name. Whitespace and special characters are stripped and the name will be lowercase.

This will create a directory with the name of the new brick inside the `bricks` folder of the Frontend Workspace. If the folder does not exist, it will automatically be created.

> **NOTE**
>
> Please note, that this script will not create new bricks in the `lib/bricks` folder, but in `/bricks` to ensure the `lib` folder stays untouched and ensure smooth upgrades of the frontend workspace.

The `create-brick` command will generate the initial brick structure as described in Section 4.3, "Bricks Structure" [39] and creates the following files, which contain basic examples of a brick's core functionalities:

Configuration files

The Hello-World brick contains different configuration files. The most important one is the `package.json`. The Prettier *`scripts`* and *`devDependencies`* are already predefined in said config file, while the *jQuery* and *js-logger* dependencies are still commented out. Move these entries to *`dependencies`* to activate them. They will be used in this Brick's JavaScript.

There are also two JavaScript file entries in the `package.json`. These files are described further below. While `index.js` is the primary entry point, that can be used by other package (for example, your theme), the `init.js` will be called initially when the brick is loaded. Learn more about how the JavaScript entry point works in Section 4.3, "Bricks Structure" [39].

The Prettier files `.prettierignore` and `.prettierrc` are configuration files for Prettier code formatter. While `.prettierrc` contains rules on how to format the brick's code, `.prettierignore` excludes folders and files from formatting. Visit https://prettier.io/ to learn more about Prettier. If you don't want to use Prettier, simply delete these configuration files and remove the prettier scripts and devDependencies entries in the `package.json`.

JavaScript files

You can find 3 different JavaScript files in `/src/js`. As mentioned before, `index.js` serves as the primary entry point to this brick. You should use this file to export all JavaScript functionality you want to share with other packages. It currently only exports the functionality of `<brickName>.js` file, but could also export any other js file you create. The `init.js` file should be used to execute code as soon as the brick is loaded. Right now, nothing happens when the brick is loaded. To make the example function in `<brickName>.js` work, simply uncomment the code in this file and in `init.js`. And don't forget to activate the required dependencies *jquery* and *@coremedia/js-logger* in the `package.json`. Starting your theme with this brick enabled, should now display a "Brick <brickName> is used." output in your browser's console.

```
"main": "src/js/index.js",
"coremedia": {
  "type": "brick",
  "init": "src/js/init.js"
}
```

Example 5.1. Example configuration in `package.json` for a brick

Localization

The Hello-World brick comes with 2 localization files: `src/l10n/<brick Name>_de.properties` and `src/l10n/<brickName>_en.proper`

`ties`. There is already an entry in each of these files, which localizes a simple welcome-Text key. This key is used in the example `Page._body.ftl` template. See Section 4.6, "Localization" [46] to learn more about localization.

SCSS files

The Hello-World brick generates a `src/sass/_partials.scss` and `src/sass/_variables.scss` as entry points for the brick's SASS files. All other SCSS files should be imported in one of these files, depending on whether they contain CSS rules or variable declarations. You will find one example variable in `src/sass/variables/_<brickName>.scss` and a CSS rule, that makes use of this variable in `src/sass/partials/_<brickName>.scss`. See Section 4.4, "Sass Files" [42] to learn more about how variables and partials are separated in the frontend workspace.

Templates

The Hello-World brick comes with just a single template: `src/templates/com.coremedia.blueprint.common.contentbeans/Page._body.ftl`. This template renders the localized "Hello World" string instead of everything else your theme comes with, except your theme contains an own `Page._body.ftl` file, which would override this one. After making sure, the new Brick works and is included correctly in your theme, you should remove this template to be able to render the real contents of your page. See Section 4.8, "Templates" [53] to learn more about the usage of templates.

You´ll need to run `pnpm install` from the root of the frontend workspace to install the dependencies of the new brick before the brick can be used.

To use the created brick, you will have to install the brick in a theme as described in Section 5.3, "Using Bricks" [67].

5.3 Using Bricks

In order to use a brick, you need to adjust your theme accordingly. This includes adjusting your theme's `package.json`, using a so called "smart-import" mechanism from your SCSS files and adding a resource bundle for all bricks to the theme.

> CAUTION
>
> To keep the bricks maintainable and easy to upgrade it is highly recommended to make no changes to the files and folders in the brick directory, except creating your own brick. Otherwise, upgrading via a patch file may no longer be possible.

Installing a brick

First of all, the brick needs to be added to your theme's dependency list. This can be done using the shell:

```
cd themes/<name>
pnpm add @coremedia/brick-media@^1.0.0
```

This will install the brick and all its dependencies (which might also be bricks) in your `node_modules` folder and add it to the "dependencies" list of your theme's package.json. The order in which bricks are installed does not matter.

Activating a brick

Most parts of a brick just like templates and initializing JavaScript code will automatically be included in the theme build after installing a brick. Because of technical reasons this automation needs to be added for SCSS and resource bundles when adding the first brick. For all further bricks no additional adjustments need to be made.

> CAUTION
>
> A brick always assumes that all of its parts are activated. Activating only parts (for example only JavaScript, not its styles) of a brick is not intended. CoreMedia strongly suggests considering this when using the brick so future updates to the Frontend Workspace or bricks will not break the theme.

Including SCSS code

Including the SCSS code of all brick dependencies is handled by using the "smart-import" mechanism. As explained in Section 4.4, “Sass Files” [42] SCSS code is separated into variables and partials. The variables of all bricks need to be included before any partial. Adjustments to the variables of a brick need to be made even before that. A usage of the "smart-import" mechanism in the SCSS entry file of a theme (`src/sass/$theme-name.scss`) looks like this:

```
...
// Dependency variables
@import "?smart-import-variables";
...
@import "?smart-import-partials";
...
```

Including Resource Bundles

The resource bundles of all bricks are aggregated and merged into a "Bricks" resource bundle. CoreMedia Bricks include English and German by default. English is the master resource file language. In case you are including any bricks, you need to add it to the `l10n.bundleNames` entry of the Section 6.2, “Theme Config” [117]. For more information about localization and resource bundles read Section 4.6, “Localization” [46].

```
<resourceBundles>
  <resourceBundle>l10n/Bricks_en.properties</resourceBundle>
</resourceBundles>
```

> **NOTE**
>
> Make sure that the theme's resource bundle is always the first entry so you can override any localization provided by the bricks with your own.

```
<resourceBundles>
  <resourceBundle>l10n/ThemeName_en.properties</resourceBundle>
  <resourceBundle>l10n/Bricks_en.properties</resourceBundle>
</resourceBundles>
```

Example 5.2. Example of a typical `resourceBundles` property of a theme

5.4 Using an Example Brick

As described in Section 4.3, "Bricks Structure" [39] example bricks are not intended to be used directly as they are subject to change without a clear upgrade path. Instead of that CoreMedia advises to create a copy of the example brick you want to use. This approach will be referred to with the term `eject` as basically you will eject the brick from our delivered packages. This section describes how to achieve this manually or by using our command line tool.

Manual Approach

All example bricks are located in the `bricks` folder of the Frontend Workspace and are prefixed with `example-`.

1. Find the brick you want to copy.
2. Create a copy of the whole brick folder except for `node_modules` and put it into `bricks` again. You can name the folder freely, but CoreMedia advises to name it after the brick removing the `example-`.
3. Open the `package.json` of your copy.
4. Change the entry `name` to a different unique name that does not start with `@coremedia-examples/`. This is the actual name of your brick and it does not need to equal the folder name it is contained in.
5. Check the entry `dependencies`. If the brick depends on other example bricks, you need to check the next section.
6. Finally, use `pnpm install` to install the newly created bricks. Please check Section 5.3, "Using Bricks" [67] to use the newly ejected brick in your theme.

Advanced Steps for Example Brick Dependencies

If the example brick you are trying to eject has dependencies to other example bricks, you need to perform the following steps for each of them:

1. Eject the example brick dependency as described above.
2. Perform a full text search on the dependent brick and search for the old name of the dependency. Depending on the file type (JavaScript, SCSS, FreeMarker Template) you will need to adjust the corresponding usages to use your ejected brick again.

Command Line Tool Approach

There is also a command line tool that will cover most of the manual steps. Just run the following command from the root of the Frontend Workspace:

```
pnpm install
pnpm eject
```

The interactive CLI will lead you through the different steps by asking which example bricks to be ejected. It can eject multiple bricks at once and offers to also eject example brick dependencies if required. In case you already have ejected the dependencies of an example brick (in a previous usage or manually, for instance) you can also pick which of your bricks represents the ejected example brick.

CAUTION

The tool will rewrite the `package.json` including the `dependencies` entry but it will not rewrite any imports or usages in JavaScript, SCSS or FreeMarker Templates. You will need to do this manually by performing a full-text search in all files. This step however is only required by very few example bricks.

5.5 Theme Inheritance

When creating a new theme, you can choose to start from scratch or derive from another theme and make use of all resources and files, located in the parent theme. Your new child theme may then extend the parent theme by adding more dependencies, templates, styles etc. However, there are certain limitations and requirements: You can only derive from one theme and this parent has to have the correct configuration and file structure. Have a look at the prerequisites to learn about the requirements to parent themes.

Prerequisites

To be able to inherit from another theme, you will have to make sure this theme meets certain prerequisites:

- The SCSS files of the parent theme should be created like shown in Example 4.3, “ Filesystem structure of a theme ” [35]. The theme needs a `src/sass/_partials.scss` and `src/sass/_variables.scss` file, as well as the `src/sass/themename.scss` file. While the latter file is simply importing the other ones,

  ```
  @import "variables";
  @import "partials";
  ```

  the `_partials.scss` will import all local partial SCSS files in the parent theme,

  ```
  // Dependency styles
  @import "?smart-import-partials";

  // Own partials
  @import "partials/example";
  ...
  ```

  and `src/sass/_variables.scss` will do the same for local variable files:

  ```
  @import "variables/example";
  ...

  // Dependency variables
  @import "?smart-import-variables";
  ```

  Please note, that the order of the imports is important and should not be changed.
- You will also have to make sure, that an `init` entry exists in the parent theme `package.json`. This entry should link to the JavaScript entry point of the theme:

  ```
  "coremedia": {
      ...
  ```

```
    "init": "src/js/<name>.js"
  },
```

How to extend the parent theme

If you choose not to use the theme creator, you will have to enable the inheritance by adding the parent theme to the list of dependencies in your `package.json` like shown in the example below:

```
cd themes/<name>
pnpm add @coremedia/<parent-name>@^1.0.0
```

You also need to adjust your `webpack.config.js` to set the `webpackConfig` correctly:

```
const webpackConfig = require("@coremedia/<parent-name>/webpack.config.js");
```

No matter if you chose to use the theme creator or add the dependency to the parent theme manually, you will have to adjust the `preview.scss` in your child theme in order to make the studio preview work correctly. CoreMedia recommends copying the `preview.scss` from the parent theme into the child theme and change the paths to the imported files accordingly like shown in the example below:

```
// Dependency variables
@import "~@coremedia/<parent-theme>/src/sass/variables/bootstrap_variables";
@import "~@coremedia/<parent-theme>/src/sass/variables/variables";

// Dependency variables
@import "?smart-import-variables";

// ### PARTIALS ###

// Dependency partials
@import "?smart-import-partials";

// Theme partials
@import "~@coremedia/<parent-theme>/src/sass/partials/preview";
```

5.6 Importing Themes into the Repository

Importing Themes

CoreMedia supports different ways to import Themes into the content repository. You can use the command line tools, *CoreMedia Studio*, or a pnpm command, described in the following sections. For a description how to use the command line tools, see Section 5.4.24, “Theme Importer” in *Blueprint Developer Manual* .

Using pnpm

Running `pnpm run deploy` inside a theme folder builds the theme and uploads it to the `/Themes` folder in the content repository. You need a valid API key, otherwise you need to login like in the web developer workflow. You also need write access to the `/Themes` folder.

Using Studio

To import a previously built theme (run `pnpm build`) into the content repository use the upload feature of the Studio Library. Go to the `Themes` directory, click on the upload icon in the toolbar of the Studio Library and select the Zip file of the theme you want to import.

NOTE

Make sure that you selected the `Themes` directory as target path in the upload dialog. Otherwise, you won't be able to select the theme as the associated theme.

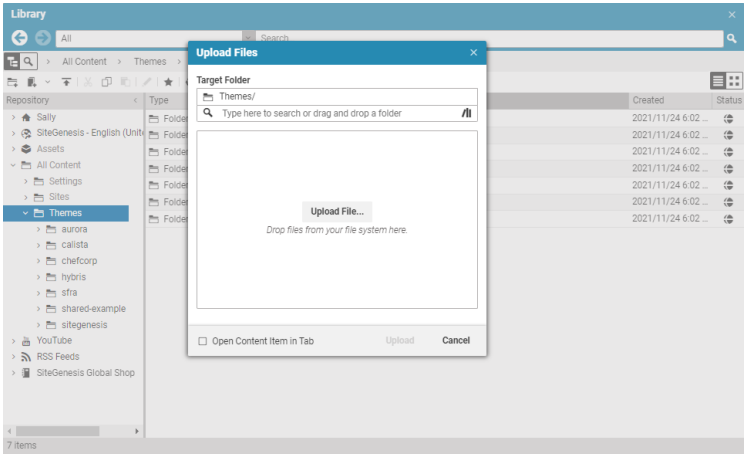

Figure 5.1. File Upload in Studio

Afterwards, select the imported theme as the associated theme for the page content of your site.

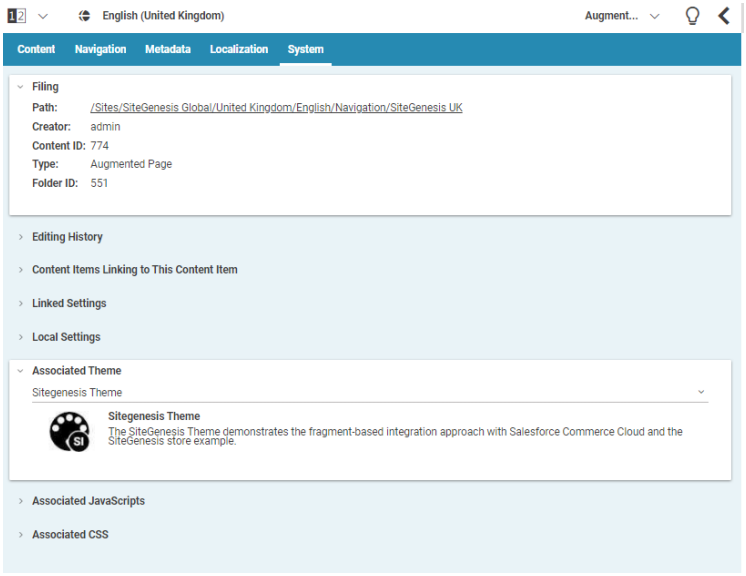

Figure 5.2. Associated Theme

Publishing the Content

Just like every other content, web resources imported to the *Content Server* need to be published in order to let the changes affect the live *CAE*. See the Studio User Manual for details about publishing content.

5.7 Referencing a Static Theme Resource in FreeMarker

Blueprint provides the FreeMarker function `bp.getLinkToThemeResource(path)` that allows creating links to static resources of the Frontend Workspace. A file is referenced by its path which needs to be specified relative to the target directory of the theme [see Section 4.2, "Theme Structure" [35]].

For example, the following snippet of a FreeMarker template creates an HTML `img` tag pointing to an image located in the theme's target folder at `img/logo.jpg`:

```
<img src="${bp.getLinkToThemeResource('img/logo.jpg')}"/>
```

NOTE

Do not move the files uploaded by the theme importer to other locations in the content repository. The paths in the FreeMarker templates would be not valid anymore and the website could be broken without even noticing it.

In order to prevent access to resources outside of the theme, the path must not contain descending path segments [".."].

CAUTION

`bp.getLinkToThemeResource(path)` is intended to be used within templates of themes and not within templates of bricks. The provided path contains knowledge about how a theme is build which may vary from theme to theme depending on the adjustments that were made to the build configuration.

5.8 Embedding a favicon in FreeMarker

You create a greater recognition value for your website by using a favicon and a touch icon. The Shared-Example Theme overwrites the new partial favicon template to add static favicon images to the page head in `Page._favicon.ftl`. To support most platforms with their own design requirements CoreMedia's example code is generated by RealFaviconGenerator.

The Blueprint FreeMarker API provides the method `bp.getLinkToThemeResource(path)` to retrieve the static URL image path. See Section 5.7, "Referencing a Static Theme Resource in FreeMarker" [76] to learn more about referencing static theme resources.

5.9 Customizing the Webpack Configuration of a Theme

You can customize the webpack configuration of a theme by editing its `webpack.config.js` which should look like this if you have not made any adjustments yet:

```
const { webpackConfig } = require("@coremedia/theme-utils");

module.exports = (env, argv) => {
  const config = webpackConfig(env, argv);

  // ...

  return config;
};
```

The imported method `webpackConfig` will generate our default configuration provided by the package `@coremedia/theme-utils`. You can simply extend this configuration by modifying the JavaScript Object that is returned by the function. This following example shows how to copy additional files:

```
const CopyWebpackPlugin = require("copy-webpack-plugin");
const path = require("path");
const { webpackConfig } = require("@coremedia/theme-utils");

module.exports = (env, argv) => {
  const config = webpackConfig(env, argv);

  // make sure that configuration for plugins exists
  config.plugins = config.plugins || [];

  config.plugins.push(
    new CopyWebpackPlugin({
    patterns: [
      {
        from: path.resolve("src/additional"),
        to: "additional",
        force: true,
        cacheTransform: true,
      }
    ]
  });

  return config;
};
```

This will copy all files located in your themes `src/additional` folder to the `additional` folder inside theme's target folder when the theme is build and also causes webpack to track changes to the files when using the Chapter 3, *Web Development Workflow* [19].

You can find more information about the configuration by checking the Webpack Documentation. More information about the CopyWebpackPlugin can be found here.

NOTE

If you do not want to use any of CoreMedia's preconfigured webpack configuration, you can remove the call to `@coremedia/theme-utils` (not recommended) and just start with an empty JavaScript Object `{}`. In that case you are starting from scratch and need to configure webpack yourself to provide a proper theme structure that can be used by the theme-importer.

5.10 Building Additional CSS Files from SCSS

As described in Section 5.9, “Customizing the Webpack Configuration of a Theme” [78], you can customize our default configuration of webpack in your theme's `webpack.config.js`. One usage might be that you need to split your themes styling into multiple CSS files in the target folder. Using our default configuration only `src/sass/$theme-name.scss` and `src/sass/preview.scss` will be compiled and saved into `target/.../css/$theme-name.css` and `target/.../css/preview.css`.

You can specify additional SCSS files - so called entry points - using the following code in the `webpack.config.js` of the theme:

```
const path = require("path");
const { webpackConfig } = require("@coremedia/theme-utils");

module.exports = (env, argv) => {
  const config = webpackConfig(env, argv);

  // make sure that configuration for entry exists
  config.entry = config.entry || {};
  config.entry["additional-styles"] = [
path.resolve("src/sass/additional-styles.scss") ];

  return config;
};
```

The example will compile the given `src/sass/additional-styles.scss` into `target/.../css/additional-styles.css`.

NOTE

There are limitations for this approach: Because of how our default webpack configuration is set up the additional CSS files can only be generated into the same folder as the other CSS files of the theme as the rewriting of `url` statements inside the SCSS code will not work properly.

5.11 Customizing the Babel Configuration of a Theme

Babel is used to compile ECMA Script of the Frontend Workspace into a form that can be understood by all browsers a theme should support [see Section 4.11, “Browser Support” [59]].

If you need to customize our default babel configuration you can to this the same way as Section 5.9, “Customizing the Webpack Configuration of a Theme” [78] but in this case you need to add a file named `babel.config.js` to the theme's root folder.

```
const { babelConfig } = require("@coremedia/theme-utils");

module.exports = api => {
  const config = babelConfig(api);

  // ...

  return config;
};
```

The imported method `babelConfig` will generate our default configuration provided by the package `@coremedia/theme-utils`. You can simply extend this configuration by modifying the JavaScript Object that is returned by the function.

You can find more information about the configuration by checking the Babel Documentation.

5.12 Embedding Small Images in CSS

The default theme build process will automatically embed images using data URLs if they are smaller than 10000 bytes. This is a feature of the url-loader used in our Webpack configuration to optimize the loading time of the website.

You can change the threshold for embedding images if this does not fit your needs by adding a build config to either the Section 6.2, "Theme Config" [117] (preferred) or the `coremedia` entry of your theme's `package.json`:

```
{
  ...
  "buildConfig": {
    "imageEmbedThreshold": 20000
  }
  ...
}
```

The example will set the threshold to `20000` bytes. Setting the value to `0` means that all images will be embedded regardless of their size (not recommended), `-1` will disable the functionality completely so images will not be embedded using a data URL.

5.13 Integrating Non-Modular JavaScript

Not all JavaScript files found via the NPM registry are written with a JavaScript Module System in mind. This might also apply to your own JavaScript if you are coming from an older CoreMedia version. One of the main differences between modular and non-modular JavaScript is the scope of the declared variables. While the latter has full access to global variables modular JavaScript will never work on the global scope but will import other modules if functionality is needed and exports its own functionality that can be reused by other modules explicitly.

First of all, CoreMedia recommends to actually migrate non-modular JavaScript into modular JavaScript, preferable by using the ES6 module system. This makes sure that all JavaScripts are up-to-date and you can use the advantages of the module system. IDEs like IntelliJ IDEA offer very good code assistance to efficiently work with the module systems.

However, in some cases migration is not possible because the JavaScript comes from a third-party library and may not be changed or it is too expensive to perform a full migration of the existing code base. In both cases the suggested approach is to use a mechanism called `Shimming` which basically means wrapping the JavaScript into an adapter that - from the perspective of the module system - makes sure that the JavaScript can be used as a module but - from the perspective of the JavaScript file - provides access to all global variables the file is operating on.

Shimming

Shimming is build into the theme build mechanism based on Webpack's *imports-loader* and *exports-loader*. Make sure you have read the basic concepts described in the Webpack Documentation.

Let's imaging there is a third-party JavaScript `./src/vendor/specialCalc.js` that expects jQuery to be found under a global variable named `jQuery`. It will perform some calculations and stores its result in a global variable named `calculationResult`. This JavaScript file may not be touched because the author doesn't think it is a good idea to use a modular system but still provides updates regularly to improve the algorithm of the calculation. To integrate this JavaScript file into a modular system you need to shim it. This can be achieved in two ways.

Shimming via Webpack Configuration

In a theme you can directly adjust your `webpack.config.js` to add configuration for shimming:

```
const path = require("path");
const { webpackConfig } = require("@coremedia/theme-utils/webpack.config.js");

module.exports = (env, argv) => {
  const config = webpackConfig(env, argv);

  config.module = config.module || {};
  config.module.rules = config.module.rules || [];
  config.module.rules.push({
    test: path.resolve("./src/vendor/specialCalc.js"),
    use: ["imports-loader?jQuery=jquery",
"exports-loader?result=calculationResult"],
  });

  return config;
};
```

Example 5.3. Shimming in webpack.config.js

This means that whenever `./src/vendor/specialCalc.js` is imported by a JavaScript module the module known as `jquery` will be provided under a variable named `jQuery`. After the script has run a variable named `calculationResult` will be exported under the name `result`.

Basically the mechanism will add code to the beginning and to the end of the JavaScript file during theme build, so the resulting output looks like this and can be used as a JavaScript module:

```
import jQuery from "jquery";

    ... (the original code of specialCalc.js) ...

    export { calculationResult as result };
```

Example 5.4. The added code

Shimming via package.json

You can also add a `shim` configuration in your theme configuration. This also works in bricks so they can provide their required shimming configuration self-contained:

```
{
  ...
  "coremedia": {
    "type: "theme",
    ...
    "shim": {
      "./src/vendor/jquery.bcSwipe": {
        "imports": {
          "jQuery": "jquery"
        },
        "exports": {
          "": "jQuery"
        }
      }
    }
```

```
    }
}
```

Example 5.5. Shimming in the theme's package.json

This will lead to the same result as the other example.

WARNING

Although it is also possible to shim a module on the fly during a `require` statement directly in the JavaScript that wants to import a non-modular JavaScript file CoreMedia does not recommend using it. The syntax is hard to read but more important it will break the externals configuration as modules are imported although they are marked as external dependency.

5.14 Changing the pnpm Registry

Sometimes it might be necessary to adjust from which source pnpm will download its packages (for example if you want to use a mirror or the original registry cannot be reached from your location). pnpm also supports different registries for specified scopes.

In general this can be achieved utilizing the command line via `pnpm config set` or by directly making changes to the `.npmrc` file. For more details take a look at the official pnpm documentation.

5.15 Rendering Markup

In CoreMedia Markup is rendered with @cm.include of an Object that contains Markup (for example, CMArticle)

```
<@cm.include self=self.detailText!cm.UNDEFINED />
```

NOTE

If content is embedded in Markup, it can be rendered by a template with the view `asRichtextEmbed`. (for example, embed `CMPicture` in the Markup, the Template would be `CMPicture.asRichtextEmbed.ftl`).

5.16 Rendering Container Layouts

There are various ways container layouts can be rendered. This section will describe the approach CoreMedia is using in the example banner bricks and themes. For this, you should have understood the basic concepts of view dispatching (see the Content Application Developer Manual for more details).

Definition

In CoreMedia's bricks and themes a container layout is a visual component that consists of a header and a grid. It is based on the model `com.coremedia.blueprint.common.layout.Container`. The header of a container layout can contain additional information like a teaserTitle or teaserText if the information is provided. The grid will arrange the items found in the model specifically based on the type of the container layout.

Typical beans implementing the interface `com.coremedia.blueprint.common.layout.Container` are `com.coremedia.blueprint.common.layout.PageGridPlacement` and `com.coremedia.blueprint.common.contentbeans.CMCollection`. Their `viewtype` property (also referred to as `Layout Variant` in Studio) determines which type of container layout will be used.

Involved Models and Views

The following class diagram gives an overview about the different models and views that are involved in the rendering of container layouts and their relation to each other. Every method in the corresponding model stands for a Freemarker template, for example, the `asPortraitBanner` method of the model `CMTeasable` stands for `CMTeasable.asPortraitBanner.ftl`.

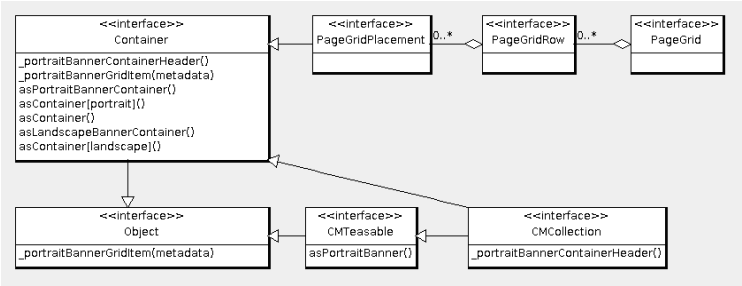

Figure 5.3. Class diagram of Models involved in Container Rendering

NOTE

The package names of the corresponding models are omitted.

Although the interface `Object` does not exist in Java, CoreMedia decided to keep the diagram simple by introducing it instead of adding actual implementation classes which can inherit from `java.lang.Object`. The goal is to also visualize the fallback view.

Of course much more views do exist for the corresponding models in the Frontend Workspace. Here, only the views are shown that are covered by this section.

`asPortraitBanner`

Renders the given bean as a portrait banner (see Section 6.4.14, "Example Portrait Banner" [163]).

`asPortraitContainer`

This view is used to render the outer HTML structure of the `portrait` container layout. It will utilize the partial views `_portraitBannerContainerHeader` and `_portraitBannerGridItem`.

`_portraitBannerContainerHeader`

Renders the header part of the portrait banner container based on the given bean. While a `PageGridPlacement` will not add any information for `CMCollection` the content of the `teaserTitle` is rendered.

`_portraitBannerGridItem`

Renders the given bean by including the view `asPortraitBanner` and wrapping it into an HTML structure that is needed to render it as a grid item.

The portrait banner grid does not support nested container layouts so whenever a `com.coremedia.blueprint.common.layout.Container` is encountered its items will be rendered as if they are part of the outer portrait banner container.

`asContainer[portrait]`

This view is just used for dispatching the `viewtype` with the id `portrait` to the view `asPortraitContainer`.

`asContainer`

This view is included if the `viewtype` property is not set or there is no view `asContainer[id]` handling the selected viewtype `id`. So it acts as the default and also as a fallback.

As the definition of default might differ from one theme to another it is not contained in any banner brick but is located in CoreMedia's themes.

Using Container Layouts for the PageGrid

Let's see the container layouts in action. You will see the flexibility of the container layouts by using them in a `PageGrid`. You will be able to explicitly set the `viewtype` for a `PageGridPlacement` to render all its items in the portrait banner container layout or you can keep the default `viewtype` for the `PageGridPlacement` and add `CMCollection`s which set the viewtype.

To have more variety, the `landscape banner` brick (see Section 6.4.10, "Example Landscape Banner" [153]) will also be used.

NOTE

The example will not cover how to render beans not implementing the `com.coremedia.blueprint.common.layout.Container` interface inside a `PageGridPlacement` or `CMCollection` if no layout is picked. You can see one possible solution in CoreMedia's example themes (`shared-example-theme`, for instance).

The approach will render non-container items as `left-right banners` that do not require a surrounding container layout to work. This also allows mixing container and non-container items in a single `PageGridPlacement` or `CMCollection`.

Preparation

If you want to replay the example you need to do the following things in the Frontend Workspace:

1. Create a new theme and add the bricks `@coremedia/brick-page`, `@coremedia-examples/brick-portrait-banner` and `@coremedia-examples/brick-landscape-banner` (see Section 5.1, "Creating a New Theme" [62]).
2. Create a template `src/templates/com.coremedia.blueprint.common.layout/Container.asContainer.ftl` with the following content:

```
<#-- @ftlvariable name="self"
type="com.coremedia.blueprint.common.layout.Container" -->

<#list self.items![] as item>
  <@cm.include self=item view="asContainer" />
</#list>
```

Example 5.6. Container.asContainer.ftl

3. Create a template `src/templates/com.coremedia.blueprint.common.layout/PageGridPlacement.ftl` with the following content:

```
<#-- @ftlvariable name="self"
type="com.coremedia.blueprint.common.layout.PageGridPlacement" -->

<div id="cm-placement-${self.name!""}"
class="cm-placement"<@preview.metadata
data=[bp.getPlacementPropertyName(self),
bp.getPlacementHighlightingMetaData(self)!""]/>>

  <@cm.include self=self view="asContainer" />
</div>
```

Example 5.7. PageGridPlacement.ftl

4. (optional) To make images work you need to add responsive-image settings `src/settings/Responsive Images.settings.json` which could look like this to enable the crops used by the banner types:

```
{
  "enableRetinaImages": false,
  "responsiveImageSettings": {
    "portrait_ratio1x1": {
      "widthRatio": 1,
      "heightRatio": 1,
      "0": {
        "width": 300,
        "height": 300
      }
    },
    "portrait_ratio2x3": {
      "widthRatio": 2,
      "heightRatio": 3,
      "0": {
        "width": 300,
        "height": 450
      }
    },
    "landscape_ratio16x9": {
      "widthRatio": 16,
      "heightRatio": 9,
      "0": {
        "width": 480,
        "height": 270
      }
    }
  }
}
```

Example 5.8. Responsive Images.settings.json

5. (optional) To adjust the MIME type / file extension of links to image variants add a property `linkMimeTypeMapping` for example next to `responsiveImage Settings` in the example above. The property configures a mapping of original image MIME type to the MIME type that should be used when building links to the variants. The respective JSON to map for example `image/jpeg` to `image/png` could look like this:

```
{
  "linkMimeTypeMapping": {
    "image/jpeg": "image/png"
  },
  "responsiveImageSettings": {...}
  ...
}
```

For details see Section 5.4.14, "Images" in *Blueprint Developer Manual* .

6. (optional) To have property responsive styling edit `src/sass/_variables.scss`:

```
// Own variables first
$cm-screen-sm-min: 800px;
$cm-screen-lg-min: 1200px;

$breakpoints: (
  "xs": "screen and (max-width: #{$cm-screen-sm-min - 1})",
  "xs-and-up": "screen and (min-width: 0)",
  "sm": "screen and (min-width: #{$cm-screen-sm-min}) and (max-width:
#{$cm-screen-lg-min - 1})",
  "sm-and-up": "screen and (min-width: #{$cm-screen-sm-min})",
  "lg": "screen and (min-width: #{$cm-screen-lg-min})",
  "lg-and-up": "screen and (min-width: #{$cm-screen-lg-min})",
  "pt": "print"
) !default;

// Dependency variables
@import "?smart-import-variables";
```

Example 5.9. _variables.scss

7. Deploy the theme (see Section 5.6, "Importing Themes into the Repository" [73]).

On the content side make sure that you prepare the following content in the *CoreMedia Studio*:

1. Create a new page and use the newly uploaded theme
2. Configure a page grid with 2 placements, e.g. `placement1` and `placement2` (see Section 4.5.1.1, "Editing a Page Grid" in *Studio User Manual*)
3. Create two layout Variants for `CMChannel` below `Options/Viewtypes` and make sure to set the `Layout` field to `portrait` and `landscape` respectively (see Section 4.5.1.2, "Adding a Layout Variant" in *Studio User Manual*)
4. Assign the layout variant `portrait` to `placement1` and add some articles.
5. Assign no layout variant (Default) to `placement2` and add two collections: `collection1` and `collection2`.
6. Assign the layout variant `landscape` to `collection1` and add some articles
7. Assign the layout variant `portrait` to `collection2` and add some articles

Result

You should now see three container layouts: The first layout represents `placement1`. It has no header and all items are displayed as portrait teasers. The second and third layout represent the content of `placement2`. If the teaser title of `collection1` and `collection2` is set it will be rendered in the header. The items of the corresponding collections are rendered as defined by their layout variant (landscape and portrait).

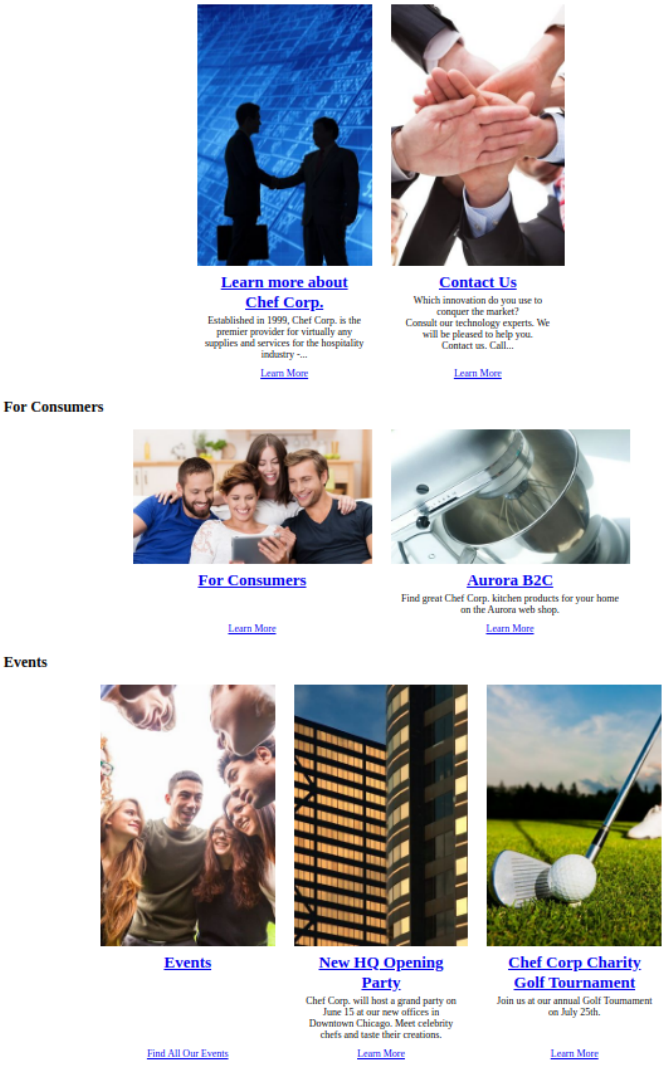

Figure 5.4. Container layouts for PageGrid

The following sequence diagrams demonstrates how each view is involved in the rendering of the page grid:

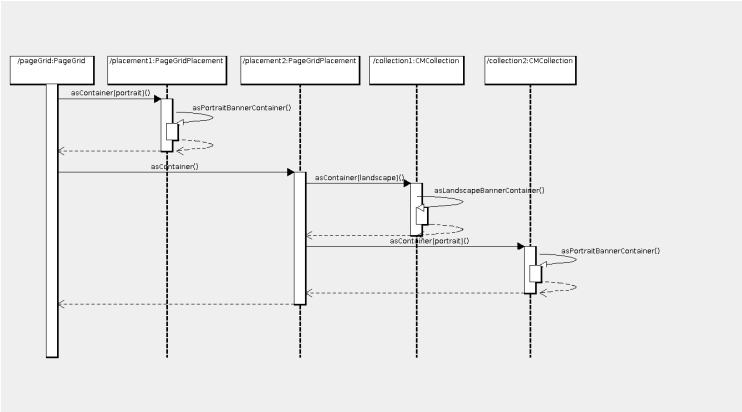

Figure 5.5. Sequence diagram showing view dispatching in the page grid

NOTE

In order to keep the diagram readable the sequence stops at `asPortraitBanner Container`. This will be covered by the next section.

Nested Collections

CoreMedia's container layouts can also handle nested collections. Let's assume instead of having multiple articles in `placement1` (see previous section) you have a single article `article1` and a collection `nestedCollection`. The later also contains an article `nestedArticle`.

The following sequence diagram shows how the rendering works in this case. The starting point is where the view `asPortraitBannerContainer` is included:

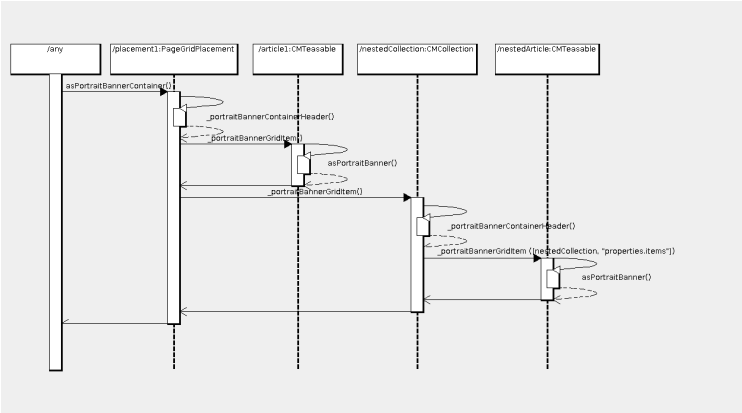

Figure 5.6. Sequence diagram showing view dispatching for nested items

As you can see the inclusion of `_portraitBannerGridItem` in `placement1` leads to another inclusion of `_portraitBannerGridItem` which also passes the metadata data of `nestedCollection` and its `items` property. This is important for the preview based editing as otherwise the hierarchical information about how the `nestedArticle` comes into the rendering (which is `placement1` -> `items` -> `nestedCollection` -> `items`) is incomplete.

5.17 Templates for HTTP Error Codes

In CoreMedia it is possible to provide templates for HTTP Error Codes.

In the Blueprint, properties are set for the HTTP Error Codes `400` and `404`. Therefore, the error codes are available as a view of `HttpError` and a template can be written for them (for example, `HttpError.404.ftl`).

`com.coremedia.objectserver.web.HttpError` is described in the "Blueprint Frontend Javadoc".

NOTE

To provide views for other HTTP Error Codes, the Spring-Configuration for the bean `blueprintHttpErrorView` has to be adapted.

5.18 Using Code Splitting for JavaScript

Webpack allows splitting your code into multiple smaller bundles - so called chunks - which can be loaded as soon as the code is required (see https://v4.webpack.js.org/guides/code-splitting/). Code that should not be included with the main bundle needs to be loaded using the dynamic module import. Assuming you have the following code:

```
import banners from "./banners";
import videoIntegration from "./videoIntegration";

banners.init();
videoIntegration.init();
```

Example 5.10. Static Import for videoIntegration

Let's say your website always has banners but only a few pages actually have videos. In this case you can use code splitting to only load the JavaScript code that initializes videos if there is a video on the current page:

```
import banners from "./banners";

banners.init();

if (document.querySelector("video")) {
  import("./videoIntegration").then(videoIntegration => {
    videoIntegration.init();
  });
}
```

Example 5.11. Dynamic Import for videoIntegration

While the `static import` statement is used to load the banners, the new `dynamic import` can be used like a function which returns a Promise (see https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise). This promise is fulfilled after the module has been loaded asynchronously. You can find more information about how to use `import` here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import.

Using dynamic imports reduces the actual code that needs to be loaded for pages that do not have videos. Usually Webpack bundles all JavaScript code used on your page into a big chunk that needs to be loaded before that page can finish loading. The videoIntegration code of the example will now be loaded asynchronously and only on demand. For this, Webpack will now create an additional chunk containing only the code for the videoIntegration (and its dependencies if they are not required by the rest of your JavaScript code).

In order to work with the CoreMedia link building we will generate a mapping for every chunk so the correct link to the corresponding code resource is being used. This is needed to keep our different development round trips working. The mapping is loaded before any other JavaScript of the theme is being loaded and defaults to `js/chunkPathById.js`. You can customize the path in your theme configuration using `buildConfig.chunkMappingPath`.

NOTE

In case you want to support older browsers like the Internet Explorer 11 you need to add a corresponding Promise polyfill to your code. We suggest adding the polyfill to every module that uses a dynamic module import.

For Section 6.3.7, "MediaElement" [132] we already make use of dynamic imports and use the package `promise-polyfill`.

5.19 Building Standalone JavaScript Files

Sometimes certain JavaScript files of a theme are meant to be included on a page without loading the whole theme. This section describes a possible solution that is supported in our theme build.

First of all you need a script that is meant to be embedded on another page. For simplicity the goal for now is to write something to the console. The script could be located at `src/console-message.js` and could look like this:

```
console.log("Standalone feature loaded!");
```

In order to include this script into your theme build you need to define a script entry in your `theme.config.json` and configure it in a special way:

```
{
  "name": "your-theme",
  ...
  "scripts": [
    {
      "type": "webpack",
      "src": "src/js/your-theme.js"
    },
    ...
    {
      "type": "webpack",
      "src": "src/js/console-message.js",
      "runtime": "console-message",
      "include": false,
      "smartImport": "console-message"
    }
  ]
  ...
}
```

In addition to the default script `src/your-theme.js` we have defined a new script entry for `src/console-message.js`. The script makes use of a couple of configuration options:

- This script utilizes the `runtime` configuration which tells the build process to isolate the script from other script files so no common runtime file is shared among them. This also means that all the libraries used by the script will be running in a separate instance.
- (optional) The `include` option is set to `false` to prevent the script from automatically being loaded when the theme is loaded.
- (optional) By setting the `smartImport` config to `"console-message"` we tell the theme build to only automatically load bricks that also have the `"console-message"` smartImport option in their `package.json` (see Section 4.1, “Structure of the Workspace” [30] and Section 6.2, “Theme Config” [117]). In this case it means that no brick is automatically loaded.

The theme can now be build and uploaded to the studio. When inspecting the generated content for the theme there will be a `CMJavaScript` content item named `console-message.js` in the theme's `js` folder. As it contains a standalone script you can directly link the content to any page regardless of its currently selected theme.

6. Reference

The following sections describe and list details of available themes and bricks and additional APIs:

6.1 Example Themes

The Frontend Workspace contains a number of example themes. Just like Section 6.4, “Example Bricks” [140] can be found in the `bricks/` all example themes be found in the `themes/` folder of the frontend workspace.

CoreMedia Blueprint currently contains the following themes for the example websites:

- Section 6.1.1, “Shared-Example Theme” [104]
- Section 6.1.2, “Chefcorp Theme” [109]
- Section 6.1.3, “Aurora Theme” [111]
- Section 6.1.4, “Calista Theme” [112]
- Section 6.1.5, “Hybris Theme” [113]
- Section 6.1.6, “Sitegenesis Theme” [114]
- Section 6.1.7, “SFRA Theme” [115]

CAUTION

All listed themes are considered to be an example which is subject to change. If you want to reuse one of our themes you should create a copy of the theme and to change the package name in its "package.json". It is also advised to change the name of the theme in the theme configuration.

All themes support the same pnpm scripts to install, build, develop and deploy themes. Run the following scripts inside a folder of a theme.

Installation

```
pnpm install
```

Building

```
pnpm build
```

Development

```
pnpm start
```

Deployment

```
pnpm run deploy
```

6.1.1 Shared-Example Theme

The Shared-Example Theme comes with a modern and minimal fully responsive design. Build on Twitter Bootstrap and our bricks. It demonstrates the capability to build localizable, multi-national, experience-driven websites.

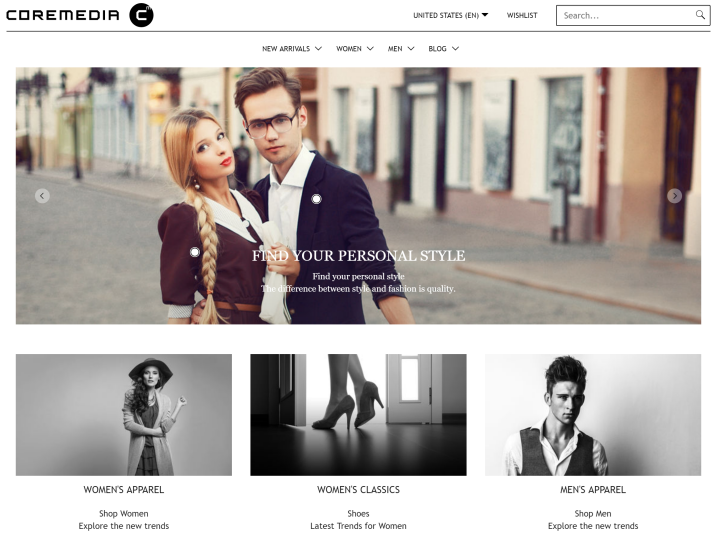

Figure 6.1. Shared-Example Theme

NOTE

This theme is the parent theme for Calista Theme and ChefCorp Theme. Read Section 5.5, "Theme Inheritance" [71] for more information about it.

CAUTION

This theme is the shared theme as foundation for the *Blueprint* themes. Even though it's a good example how to write themes, you should not use it as a base for your custom themes to avoid conflicts in the future. You should always create new themes with the **pnpm run create-theme** script, described in Section 5.1, "Creating a New Theme" [62].

Features

Favicons

The Shared-Example Theme provides embedded favicons defined in `Page._favicon.ftl`.

Responsive Page Grid

The Shared-Example theme renders the placements of a site in an own responsive page grid based on the CSS flexible box layout model (flexbox). The theme's page grid works similar to the Twitter Bootstrap's grid system and is defined in `Container.asGrid.ftl` and `_flex-grid.scss`. It can be used as follows:

```
<div class="cm-flex-row cm-flex-row--center">
  <div class="cm-flex-col-xs-6 cm-flex-col-md-2"> ... </div>
  <div class="cm-flex-col-xs-6 cm-flex-col-md-2"> ... </div>
  ...
</div>
```

The above example adds the `cm-flex-row--center` class to the row div, which displays all columns centered in the corresponding row.

The Shared-Example theme also comes with templates to render different placements of a site uniquely. The `PageGridPlacement.ftl` includes different templates for placements, that must be named "header", "footer" or "footer-navigation" in your site and therefore renders their layout different from all other placements.

Banners

The Shared-Example Theme is using Example Bricks to include different banner variants hero, portrait, landscape, square, left-right and carousel.

All other items in placements are rendered as teasers in `Container.asContainer.ftl`.

Layout

The Shared-Example theme makes use of the footer brick to display the placements footer and footer navigation. It also uses the navigation brick to enable a navigation section below the header and inside the mobile header menu.

Elastic Social

The *Elastic Social* feature supports comments for articles. See the Chapter 1, *Preface* in *Elastic Social Manual* to learn more about *Elastic Social*.

Editorial Blog

The Shared-Example theme includes authors in articles and supports author detail pages for the Editorial Blog. Authors are displayed below the article text in detail pages and above the title in the default teaser layout. It also makes use of the feature to load more items of a CMQueryList via AJAX. The blog pages and author detail pages (for related items) show three items and a "load more" button, if more items are available.

Search

The Shared-Example theme makes use of the search brick to display a search input field in the header of the page. After submitting his search, the user will be redirected to a search page, where he can get an overview of the results, adjust filters or alter his search term.

Hero Banner

Hero Banner

The *Hero Banner* layout variant renders a banner with great imagery. It fills the whole width of the grid on all devices. Important: Different screen orientations need different crops. On mobile devices the image format changes to 1:1. If a placement or a collection is filled with multiple teasables, these items will be rendered as carousel with arrows indicating and navigating to the previous and next items.

The appearance of banners, rendered in the hero layout variant can differ completely from the usual layout. The following table shows which content types will be enriched with additional elements or rendered as a whole other component:

Type	Appearance
Product	Renders an additional "Shop Now" button if product offers this option
HTML	Renders the plain HTML
ImageMap	Renders an ImageMap with HotZones, Popups and indicators

Table 6.1. Special Hero Banner Types

Portrait Banner

Portrait Banner

The *Portrait Banner* Layout Variant renders a simple banner that has a portrait image and text below the image. Advanced teaser management is ignored. It is intended for products.

The following render settings are different to the default settings for default banner:

- `renderTeaserText: true`
- `renderEmptyImage: false`
- `enableTeaserOverlay: false`

The appearance of banners, rendered in the portrait layout variant can vary from the usual layout. The following table shows which content types will be enriched with additional elements or rendered as a whole other component:

Type	Appearance
Product	Renders an additional "Shop Now" button if product offers this option
HTML	Renders the plain HTML
Download	Renders an additional download icon, file name and file size
Gallery	Renders the contents of the gallery as items in a new row, even if the row prior or after the gallery content is not fully filled

Table 6.2. Special Portrait Banner Types

Landscape Banner

Landscape Banner

The *Landscape Banner* Layout Variant renders a simple banner that has a landscape image and text below the image. Advanced teaser management is ignored.

The following render settings are different to the default settings for default banner:

- `renderTeaserText: true`
- `renderEmptyImage: false`
- `enableTeaserOverlay: false`

The appearance of banners, rendered in the landscape layout variant can vary from the usual layout. The following table shows which content types will be enriched with additional elements or rendered as a whole other component:

Type	Appearance
Product	Renders an additional "Shop Now" button if product offers this option

Type	Appearance
HTML	Renders the plain HTML
Download	Renders an additional download icon, file name and file size

Table 6.3. Special Landscape Banner Types

Square Banner

Square Banner

The *Square Banner* Layout Variant renders a simple banner that has a square image and text on the image. Advanced teaser management is ignored.

Left Right Banner

Left Right Banner

The Shared-Example theme provides a *Left-Right Banner* layout variant of `CMTeasable` as teaser. It renders content items each in a row with a left aligned media content followed by a right aligned text and vice versa.

The layout is defined in the template in `Container.asContainer[left-right].ftl`. They do not call an additional grid template as default. As layout variant of the bean `CMTeasable` the parameters described in Default Banner can be used. It disables the teaser overlay functionality and shows authors, a display date, empty image background and a "load more" button for `CMQueryList`.

The following render settings are different to the default settings for default banner:

- `renderTeaserOverlay: false`
- `renderAuthors: true`
- `renderDate: true`

The appearance of banners, rendered in the left-right layout variant can vary from the usual layout. The following table shows which content types will be enriched with additional elements or rendered as a whole other component:

Type	Appearance
Product	Renders additional price, offer price and a "Shop Now" button if product offers this option
HTML	Renders the plain HTML
Download	Renders an additional download icon, file name and file size
Gallery	Renders the contents of the gallery as items, like collections
ImageMap	Renders the imagemap in front of the image

Type	Appearance
Video	Renders the video as autoplayed, looped and muted inline video, if no image is available. Also renders a play button. A Click on the banner will open a large version of the video in a lightbox.
Audio	No special view
360° Spinner	No special view

Table 6.4. Special Left-Right Banner Types

Carousel Banner

Carousel Banner

The *Carousel Banner* Layout Variant renders a carousel containing banners. The banners include a portrait image and text below the image. Advanced teaser management is ignored. There are no CTAs shown.

6.1.2 Chefcorp Theme

The Chefcorp theme provides a modern, appealing, highly visual theme. It demonstrates the capability to build localizable, multi-national, non-commerce websites.

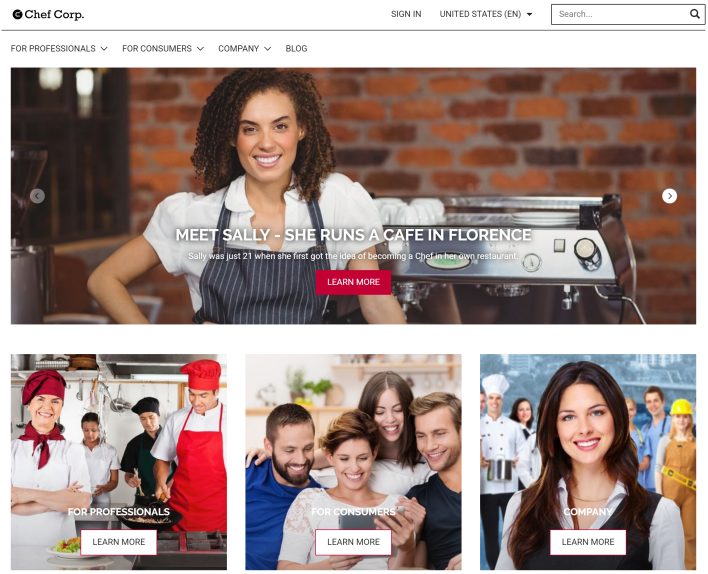

Figure 6.2. Chefcorp Theme

NOTE

This theme is a child theme derived from the Shared-Example theme. It comes with all FreeMarker templates, JavaScript, SCSS files, localizations and brick dependencies, inherited from its Parent Theme. Read Section 5.5, "Theme Inheritance" [71] for more information about it.

Features

Download Portal

A dependency to the download-portal brick enables the Download Portal features in the Shared-Example theme. An additional search field for all kinds of assets in the download portal can be used to add items to download collections and download them.

Content Catalog

The Chefcorp theme provides templates and style sheets for the content catalog. The corresponding category overview pages and product detail pages can be accessed via the Chefcorp navigation.

Elastic Social

In addition to the *Elastic Social* features, enabled in the Shared-Example theme, the Chefcorp theme does not only support anonymous commenting and reviews, but also additional *Elastic Social* features like registration, login and user management.

6.1.3 Aurora Theme

The Aurora Theme provides a modern, appealing, highly visual theme. It demonstrates the capability to build localizable, multi-national, experience-driven eCommerce websites. Integration with IBM WebSphere Commerce ships out of the box.

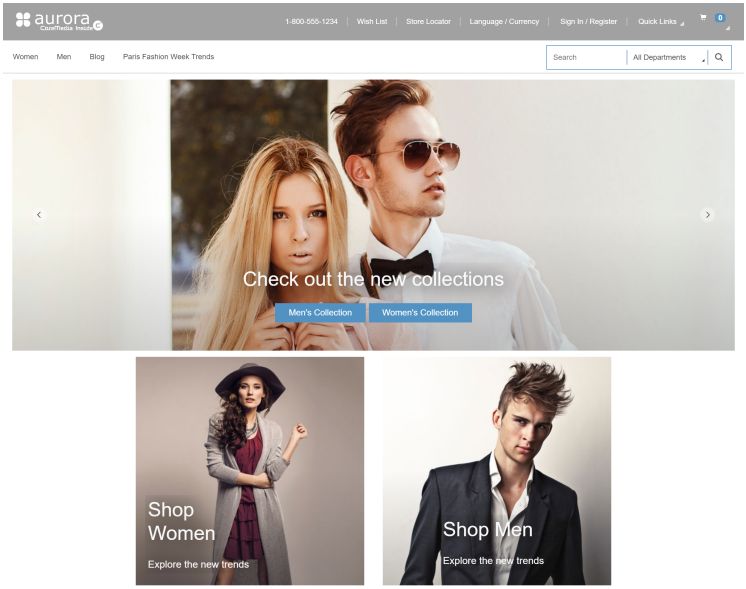

Figure 6.3. Aurora Theme

Based on a fully responsive, mobile-first design paradigm, it leverages the Bootstrap framework. It scales from mobile via tablet to desktop viewport sizes and uses the CoreMedia Adaptive and Responsive Image Framework to dynamically deliver the right image sizes in the right aspect ratios and crops for each viewport.

This theme integrates the fragment-based approach seamless into Aurora B2C store examples.

6.1.4 Calista Theme

The Calista theme comes with a modern and minimal fully responsive design. Build on Twitter Bootstrap and our bricks. It demonstrates the capability to build localizable, multi-national, experience-driven eCommerce fashion websites.

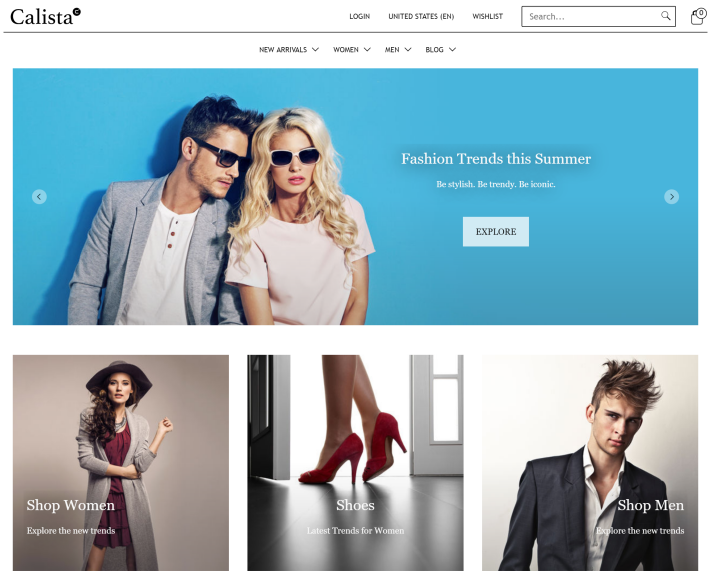

Figure 6.4. Calista Theme

The header placement provides a search field, cart icon, language chooser, a link to the login page and a section for displaying additional links next to them.

NOTE

This theme is a child theme derived from the Shared-Example theme. It comes with all FreeMarker templates, JavaScript, SCSS files, localizations and brick dependencies, inherited from its Parent Theme. Read Section 5.5, “Theme Inheritance” [71] for more information about it.

Features

eCommerce

Integration with ships out of the box. The theme is based on the Shared-Example theme (See Section 6.1.1, "Shared-Example Theme" [104]) and adds a dependency to the Example Cart Brick and the Example Product Assets.

Elastic Social

The *Elastic Social* feature is enabled in Calista by default. Commenting works in articles on the blog page, other *Elastic Social* features are not yet supported out of the box in the Calista Theme. To enable comments on other pages, these pages need to link to an *Elastic Social* settings content item in their `Linked Settings` sections. See the Chapter 1, *Preface* in *Elastic Social Manual* to learn more about *Elastic Social*.

6.1.5 Hybris Theme

The Hybris Theme provides a modern, appealing, highly visual theme. It demonstrates the capability to build localizable, multi-national, experience-driven eCommerce websites. Integration with SAP Hybris Commerce ships out of the box.

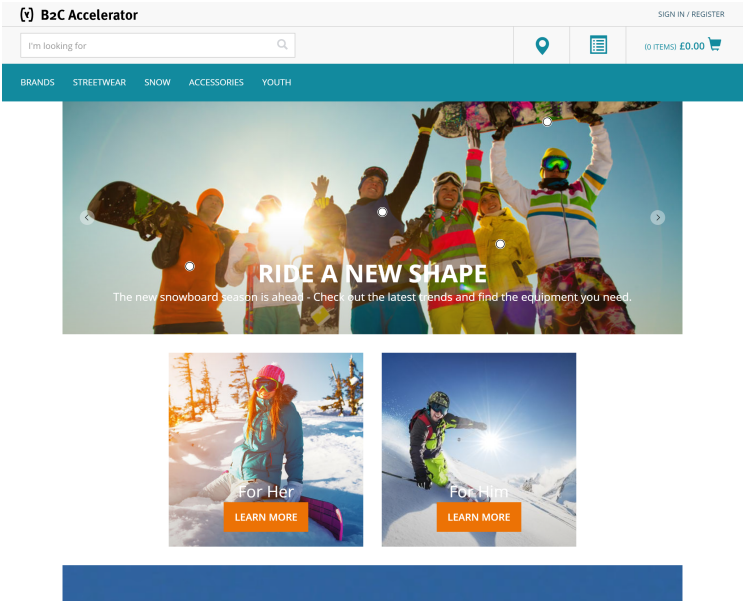

Figure 6.5. Hybris Theme

Based on a fully responsive, mobile-first design paradigm, it leverages the Bootstrap grid framework. It scales from mobile via tablet to desktop viewport sizes and uses the CoreMedia Adaptive and Responsive Image Framework to dynamically deliver the right image sizes in the right aspect ratios and crops for each viewport.

This theme integrates the fragment-based approach seamless into the SAP Hybris Apparel example.

6.1.6 Sitegenesis Theme

The Sitegenesis Theme provides a modern, appealing, highly visual theme. It demonstrates the capability to build localizable, multi-national, experience-driven eCommerce websites. Integration with Salesforce Commerce Cloud ships out of the box.

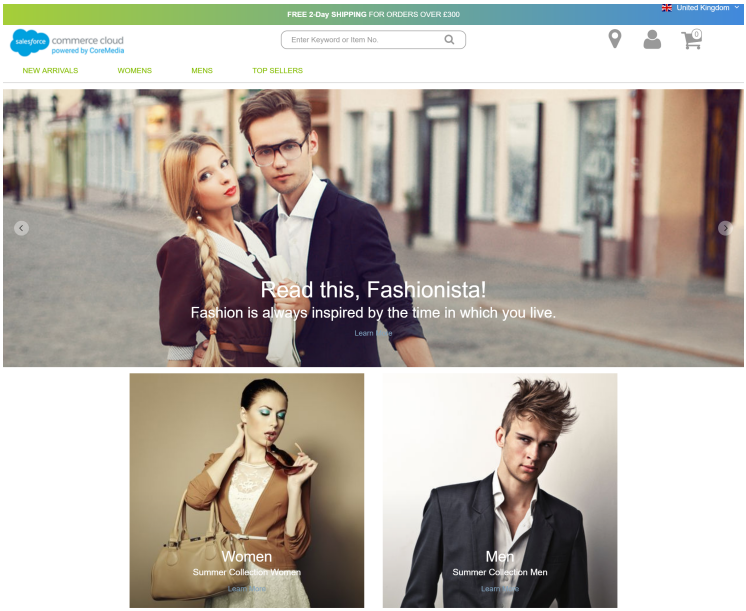

Figure 6.6. Sitegenesis Theme

Based on a fully responsive, mobile-first design paradigm, it leverages the Bootstrap grid framework. It scales from mobile via tablet to desktop viewport sizes and uses the CoreMedia Adaptive and Responsive Image Framework to dynamically deliver the right image sizes in the right aspect ratios and crops for each viewport.

This theme integrates the fragment-based approach seamlessly into the Storefront Reference Architectore.

6.1.7 SFRA Theme

The SFRA Theme provides a modern, appealing, highly visual theme. It demonstrates the capability to build localizable, multi-national, experience-driven eCommerce websites. Integration with Salesforce Commerce Cloud ships out of the box.

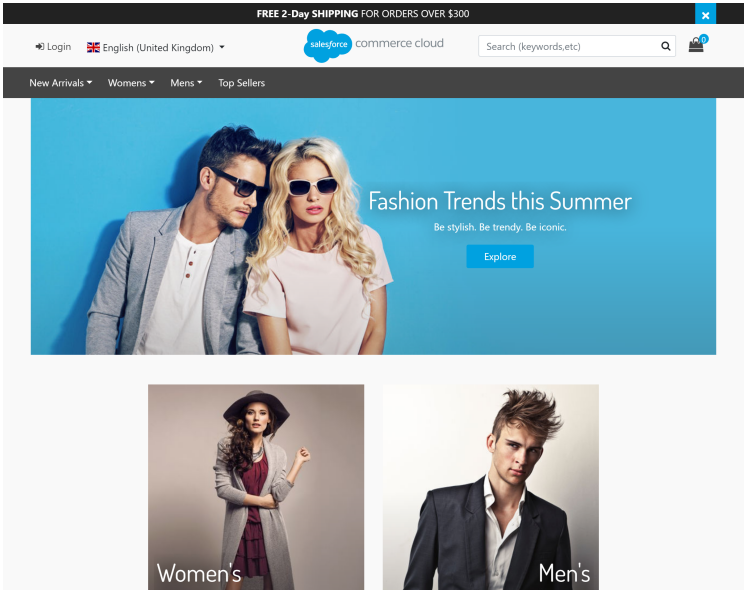

Figure 6.7. SFRA Theme

Based on a fully responsive, mobile-first design paradigm, it leverages the Bootstrap grid framework. It scales from mobile via tablet to desktop viewport sizes and uses the CoreMedia Adaptive and Responsive Image Framework to dynamically deliver the right image sizes in the right aspect ratios and crops for each viewport.

This theme integrates the fragment-based approach seamless into SiteGenesis store examples.

6.2 Theme Config

The `Theme Config` is an `JSON` file named `theme.config.json` located in the root folder of a theme. It defines meta information and build options for the theme.

Attribute	Type	Default	Description
`name`	Non-Empty String		Specifies the technical name of the theme.
`description`	Non-Empty String	`null`	The description of the theme. The first paragraph will be displayed to editors in *Studio*.
`thumbnail`	Non-Empty String	`null`	A path to a thumbnail image relative to the theme's root folder. It will be displayed to editors in *Studio*. Minimum and recommended image size is 82 x 50 pixels.
`targetPath`	Non-Empty String	`null`	Specifies the output path of the build theme. If not set the 'target' folder of the surrounding frontend workspace (or if not present the theme's) root folder will be used.
`l10n`	L10N	see below	The attribute `l10n` contains configuration for localization. Its attributes are described below.
`scripts`	Script or Array<Script>	[]	Define `script` elements which should be included in the theme here. The order of the elements also specifies the load order of the files.
`styles`	Style or Array<Style>	[]	Define `style` elements which should be included in the theme hero. The order of the elements also specifies the load order of the files.

Table 6.5. Root attributes of the theme configuration

Attribute	Type	Default	Description
`masterLanguage`	String	"en"	The language id of the master language. For further information see Section 4.6, “Localization” [46].

Attribute	Type	Default	Description
`bundleEncoding`	String	"ISO-8859-1"	The encoding of the resource bundle. Possible values are: • "ISO-8859-1" Resources are processed with "ISO-8859-1" encoding. • "UTF-8" Resources are processed with "UTF-8" encoding.
`bundleNames`	Array<N-E String>	[]	An array of non-empty strings containing the names (for example, ${bundleName}_en.properties) of resource bundles.

Table 6.6. Attributes of the `L10N` type

Attribute	Type	Default	Description
`type`	Enum		Specifies the type of the script or style. Possible values are: • "webpack" Specifies that the script or style will be build with webpack. • "copy" Specifies that the script or style will just be copied over to the target directory without any transformation. • "externalLink" The script or style is an external link.
`src`	Non-Empty String or Array<N-E String>		The source of the script or style. If `type` is set to **`"externalLink"`** the source must start with "http://", "https://" or "//" otherwise the source must match a path relative to the theme root directory.

Attribute	Type	Default	Description
`target`	Non-Empty String		The attribute only applies if `type` is set to `"copy"`. The value represents a relative path from theme's target directory to specify where the file specified in `src` should be copied to.
`entryPointName`	Non-Empty String	<calculated>	The attribute only applies if `type` is set to `"webpack"`. The value influences the base name of the generated script or style file. If it is not set it will be generated from the base name of the provided `src` attribute. If it is an array the first value will be used. Scripts will always end with `.js` while styles will always end with `.css` after the webpack build regardless of the initial type (for example, `.scss`).
`include`	Boolean	`true`	If `false` the script or style will not be included in the list of scripts or styles of the `CMTheme` content item which means that it will not be loaded automatically if you use our default templates.
`smartImport`	Non-Empty String or null	`"default"`	Bricks can define in which contexts their smart import mechanism is applied (see Section 4.1, "Structure of the Workspace" [30]). By using this config this context can be set to a different value (e.g. "preview") so only certain styles and scripts will be automatically loaded. If set to `null` all styles and scripts regardless of context will be included (not recommended).

Table 6.7. Shared attributes of the `Script` and `Style` type

Attribute	Type	Default	Description
`defer`	Boolean	`false`	If `true`, loading of the script file is deferred meaning it will be loaded on document ready.
`inHead`	Boolean	`false`	If `true`, the JavaScript file is included in the `document`'s `head` otherwise it will be loaded at the end of the `document`'s `body`.

Attribute	Type	Default	Description
`runtime`	Non-Empty String	`commons`	The attribute only applies if `type` is set to `"webpack"`. In order to reduce the size of the individual entry points and to share a single instance for ES6 modules the runtime chunk of webpack will be shared among all entry points and put into a common file. There are cases where this behavior is not desired, e.g. when embedding a JavaScript file into a website that does not load the entire theme. By changing the config to a different name the script will be bundled in a way that it will not share code and instances with scripts using a different configuration (e.g. "commons"). Important: Loading scripts with different runtimes on the same website can lead to problems if e.g. third party libraries are loaded twice which are not meant to be loaded twice.

Table 6.8. Additional attributes of the Script type

6.3 Bricks

> **CAUTION**
>
> Do not modify bricks of provided packages! This would make them way harder to maintain and upgrade! If you need to change bricks, try to overwrite it in your theme first, or at least create a new modified brick. See Section 4.3, “Bricks Structure” [39] for more detailed information.

Available Bricks

6.3.1 Default-Teaser

The default-teaser brick provides templates and basic CSS styles for default teasers. Templates exist for all kinds of `CMTeasable` and as special variants for certain other types, such as Commerce Objects, Pictures, Downloads etc.

Using the Brick

As shown in the example below, a default teaser can be displayed by including the corresponding content type with the `teaser` view. You can also pass additional CSS classes as parameters to apply custom styling to your default teasers.

```
<@cm.include self=self view="teaser"/>
```

The teaser view template works with all types and subtypes of type `com.coremedia.blueprint.common.contentbeans.CMTeasable`. The following special views exist:

- `CategoryInSite.teaser.ftl`
- `CMDownload.teaser.ftl`
- `CMGallery.teaser.ftl`
- `CMHTML.teaser.ftl`
- `CMPicture.teaser.ftl`
- `CMSpinner.teaser.ftl` (part of the 360-Spinner Brick)
- `CMTeasable.teaser.ftl`
- `LiveContextExternalChannel.teaser.ftl`
- `LiveContextProductTeasable.teaser.ftl`
- `ProductInSite.teaser.ftl`

To configure the behavior of the template you can add the following parameters to the `cm.include` tag:

Parameter	Type	Default	Description
`blockClass`	String	`"cm-teasable"`	A base name that will be used for CSS classes attached to the elements rendered by the template.
`additionalClass`	String	`""`	An additional CSS class that will be added to the outer div of the teaser.
`islast`	Boolean	`false`	Set to true to add a "is-true" CSS class to the teaser.
`renderLink`	Boolean	`true`	Per default, the whole teaser is clickable and will work as a link. Set to false to only use embedded call-to-action buttons as links.
`renderTeaserTitle`	Boolean	`true`	Whether to display the teaser title or not.
`renderTeaserText`	Boolean	`true`	Whether to display the teaser text or not.
`renderAuthors`	Boolean	`false`	Whether to display the list of linked authors. Will only be displayed if authors exist for this content.

Parameter	Type	Default	Description
`renderDate`	Boolean	`false`	Whether to display the date. Will only be displayed if a date exists for this content.
`renderEmptyImage`	Boolean	`true`	Whether to display an empty media element if no media has been linked or not.

Table 6.9. Parameters of Teasers

6.3.2 Device Detector

The `device-detector` API brick stores device and orientation information to support responsive UIs.

Technical Description

The brick provides methods to read and update device relevant information of pseudo elements at the body defined by CSS media queries.

```
import { getLastDevice } from "@coremedia/brick-device-detector";
...
  if (getLastDevice().type !== "mobile") {
    $cartPopup.toggleClass("cm-cart-popup--active");
  }
...
```

Example 6.1. Shopping Cart Example

NOTE

Please note that this brick contains `JavaScript` files, what will automatically be installed, if you add the brick to your theme `package.json`. See Section 5.3, “Using Bricks” [67] to learn how to install a brick in your theme.

6.3.3 Dynamic-Include

This brick adds support for dynamic-include functionality of the CAE to load and render a fragment from the CAE in a website and replace the placeholder DOM element. It includes templates, SCSS and JavaScript.

Using the Brick

Add the brick as a dependency to your theme. If the CAE or the content include fragments, they will automatically be loaded by this brick via JavaScript or ESI include, if supported. Even without the brick, the CAE has a simple default template `DynamicInclude.ftl` in the module `cae-base-lib`.

More information can be found in the Section 6.2.1, "Using Dynamic Fragments in HTML Responses" in *Blueprint Developer Manual* .

6.3.4 Image-Maps

The `image-maps` brick encapsulates the rendering of images, enriched with links to target pages and additional information. An editor can select areas of interest in the image and create so called Hot Zones that are used to display text overlays and link to related content. The rendering of Hot Zone indicators may depend on the layout variant of the containing collection or placement.

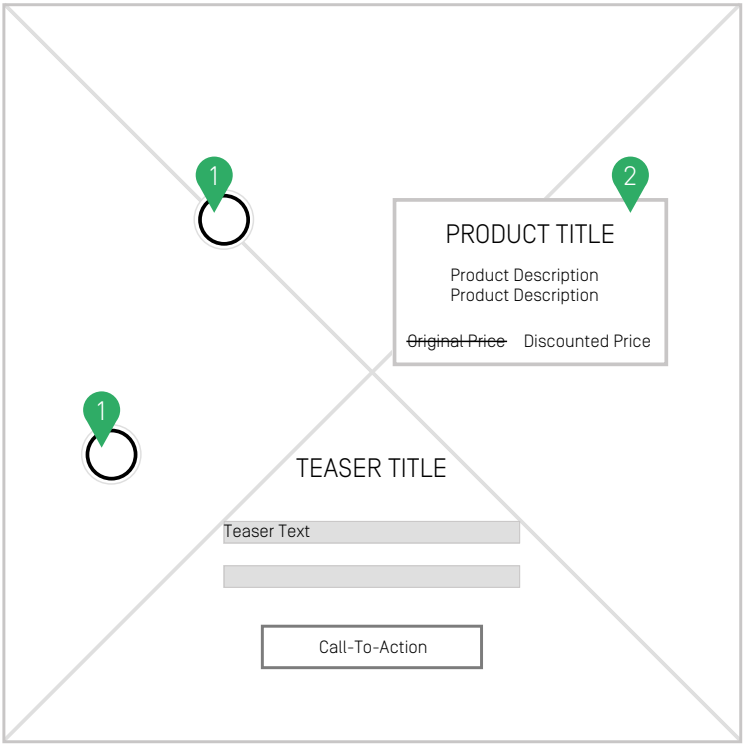

Figure 6.8. Wireframe of an image map

1. Clicking on the hot zones opens the link to a detail page. If a theme is using the brick "example popup", the target is opened in a popup instead.
2. The hot zone can be displayed as an overlay and behaves like a CTA.

Technical Description

Image Maps will work out of the box for the content type `CMImageMap` in any theme with a dependency on the `image-maps` brick. See Section 5.3, “Using Bricks” [67] to learn how to install a brick in your theme. The brick comes with the template `CMImageMap.ftl` and delegates to the detail view.

To extend the functionality of the image maps by opening the link targets in a popup overlay, the `popup` brick can be added to the theme's dependencies. For more information see Section 6.4.13, “Example Popup” [162]. For extending the image map inline

overlays otherwise simply overwrite the corresponding `*.asImageMapInlineOverlay.ftl` templates in the theme.

Dependencies

Please note that the `image-maps` brick has dependencies on *jQuery* and the *Media* brick for responsive images.

Templates and Parameters

In order to use Image Maps, you can either rely on the existing template `CMImagemap._pictureftl` or write own templates in your theme. This template renders the image with the image map.

The template can be included in your theme as follows:

```
<@cm.include self=self view="_picture" params={"blockClass": "example-class"}/>
```

To configure the behavior of the template you can add the following parameters to the `cm.include` tag:

Parameter	Type	Default	Description
`blockClass`	`String`	`""`	This will add a CSS class to elements of the image map, all beginning with the provided string.
`renderEmptyImage`	`Boolean`	`true`	If the image map should show a missing image placeholder.

Table 6.10. Parameters of the Image Map

> **NOTE**
>
> Please note that if the `image-maps` brick is not included in your theme, Image Maps will be rendered like any other `CMTeasable` for the corresponding view.

Additional Resources

- `imagemap-icon.svg`
- `imagemap-icon-hover.svg`
- `ImageMaps_de.properties`
- `ImageMaps_en.properties`

6.3.5 Magnific Popup

The `magnific-popup` API brick provides a responsive lightbox and dialog script with any device support.

Technical Description

The `magnific-popup` uses the library Magnific Popup. In combination with `node-decoration-service` and `mediaelement` it delivers a robust lightbox for video, images and text.

```
import { addNodeDecoratorByData } from
"@coremedia/brick-node-decoration-service";
import { default as magnificPopup } from "@coremedia/brick-magnific-popup";

...
addNodeDecoratorByData(
  {},
  "cm-product-assets",
    function($target) {
      const $carousel = $target.find(".cm-product-assets__carousel");
      magnificPopup($carousel, {
        gallery: { enabled: true },
        delegate: ".cm-product-asset[data-cm-product-asset-gallery-item]",
        callbacks: {
        ...
        },
      });
    }
);
```

Example 6.2. Carousel Example

NOTE

Please note that this brick contains `JavaScript` files, what will automatically be installed, if you add the brick to your theme `package.json`. See Section 5.3, “Using Bricks” [67] to learn how to install a brick in your theme.

6.3.6 Media

This brick offers the following features:

- CMPicture support with different image sizes for various viewport dimensions (responsive images). This means, that different crops of an image can be displayed on different devices.
- CMVideo support to render a native HTML5 video element.
- CMAudio support to render a native HTML5 audio element.

NOTE

To support the playback of videos from external sources like YouTube, Vimeo etc. the mediaelement brick is required. For more information visit Section 6.3.7, "MediaElement" [132]

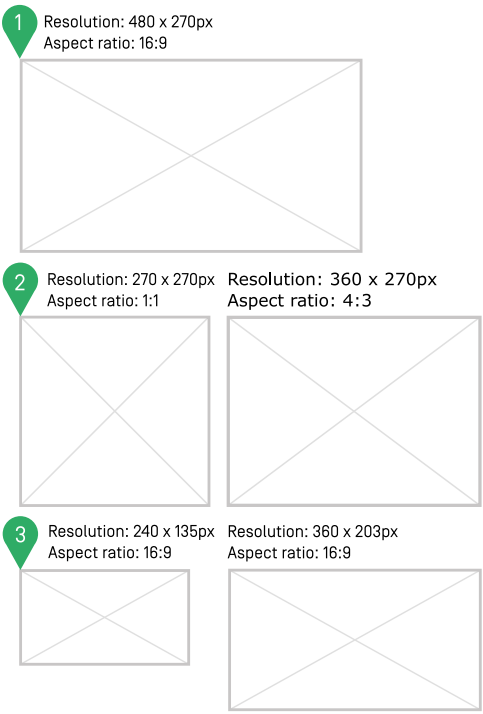

Figure 6.9. Wireframe of media

1. The image in its default size and shape
2. It can be fitted into different aspect ratios according to the parent container

3. And rendered with different resolutions for smaller use cases

Using the Brick

The brick provides a `media` view for content of type `com.coremedia.blueprint.common.contentbeans.CMPicture`, `com.coremedia.blueprint.common.contentbeans.CMVideo` and `com.coremedia.blueprint.common.contentbeans.CMAudio` so the first item in the `media` property of a `com.coremedia.blueprint.common.contentbeans.CMTeasable` could be rendered using:

```
<@cm.include self=self.firstMedia view="media" />
```

Templates and Parameters

These templates can be included in your theme (for example, in a `CMVideo.asHero.ftl` template) as follows:

```
<@cm.include self=self view="media" params={"preload": true}/>
```

Responsive Images

Images need to be available in various sizes and resolutions to fit different use cases. For example in a 4x3 aspect ratio for a teaser, 16x9 in a hero teaser and both scaled down for a mobile view as well. The media brick provides an efficient way to choose the best fitting image for any case.

At first the responsive image settings need to be configured in your sites content and linked to its settings. For more information on how to do this, configuring all image croppings and the available settings see Section 5.4.14, "Images" in *Blueprint Developer Manual* . The different image croppings you define here are then available in the frontend. When including a CMPicture in a template using the `media` view, an object containing URLs for all defined variants will be added in a `data-cm-responsive-media` attribute to the HTML `img` element.

The picture and its parent `div` element are essential units. The `img` has a CSS class consisting of the block class and a `__picture` suffix. This positions the image absolute in its parent. The parent has the same block class with a `__picture-box` suffix. This renders a `before` pseudo element responsible for the correct height ratio defined by its `padding-top` value. Therefore, CoreMedia provides the SCSS mixin `aspect-ratio-box` to receive the wanted aspect ratio.

```
@include aspect-ratio-box(4, 3);
```

NOTE

The matching crop to the values for the aspect-ratio-box must be defined in the responsive image settings.

For every page load and viewport size change the responsive image JavaScript is triggered for every image with the `cm-responsive-media` data attribute. It decides which is the best fitting image from the set of responsive images for the height and the width of the parent image-box `div` and puts its URL into the `src` attribute of the image.

The view accepts the following parameters.

Parameter	Type	Default	Description
`classBox`	`String`	`""`	CSS class for the outer div that contains the image and title.
`classMedia`	`String`	`""`	CSS class for the div containing the image.
`disableCropping`	`Boolean`	`false`	When set to true, in every case the highest available resolution of the image is used and responsive images is disabled.
`background`	`Boolean`	`false`	When set to true, the image is linked as `background-image` in the `style tag` of the block div.
`metadata`	`Array`	`[]`	Additional PDE Information to attach to the outer div of the image.
`metadataMedia`	`Array`	`[]`	Additional PDE Information to attach to the image itself.
`additionalAttr`	`Map`	`{}`	This adds attributes to the `img` tag.

Table 6.11. Parameters of the media view for responsive images

NOTE

Correctly configured responsive image settings that are linked to the site are mandatory for the responsive images function to work! The fallback is one image with its highest resolution available.

Video and Audio

To configure the behavior of the video or audio elements you can add the following parameters to the `cm.include` tag:

Parameter	Type	Default	Description
`hideControls`	`Boolean`	`false`	Hide the control panel for audio and video playback
`autoplay`	`Boolean`	`false`	The media file starts playing automatically after it has been loaded
`loop`	`Boolean`	`false`	The audio or video plays in an infinite loop
`muted`	`Boolean`	`false`	The video is muted
`preload`	`Boolean`	`false`	The browser starts loading the first part of the media file

Table 6.12. Parameters of the media brick

NOTE

Please note, that setting these parameters will overwrite the settings, defined in the content itself. A Studio user can define the `autoplay`, `loop`, `muted` and `hideControls` configuration of videos and audio files by changing them in the content form of the content. Since the Studio configuration is only used as a fallback, the configuration by template parameters will always finally decide the player's behavior.

Additional Resources

- `playicon.param.svg`
- `Video_de.properties`
- `Video_en.properties`

6.3.7 MediaElement

The mediaelement brick provides a common API (Media Element) to integrate video and audio from the CMS like HTML5 and MP3 or external videos like YouTube, Facebook or Vimeo using the CoreMedia content type "video".

Technical Description

This brick relies on MediaElements.js to provide the same API and unified experience for every type of video and audio across browsers. Therefore, it will be wrapped in a MediaElement fake DOM element.

The following external video sources are supported in our implementation by default but can be expanded:

- YouTube
- Facebook
- Vimeo

Dependencies

This brick has dependencies on the npm packages `jQuery` and `MediaElement`, some `SASS` and `JavaScript` from the frontend `lib` folder and the `media` brick.

6.3.8 Node Decoration Service

The `node-decoration-service` brick provides functionality and DOM manipulations based on events and selectors. It's intention is to support fragment scenarios to enrich pages with other content or components.

Technical Description

The `node-decoration-service` will be executed after all DOM ready functions have finished. It only accepts node decorators on selectors and data attributes based on jQuery.

```
import { addNodeDecoratorByData } from
"@coremedia/brick-node-decoration-service";

// JQuery Document Ready
$(function() {
  // add node decorator for imagemaps
```

```
  addNodeDecoratorByData({}, "cm-imagemap-popup", imageMapAsPopup);
});
```

Example 6.3. Imagemap Example

NOTE

Please note that this brick contains `JavaScript` files, what will automatically be installed, if you add the brick to your theme `package.json`. See Section 5.3, "Using Bricks" [67] to learn how to install a brick in your theme.

6.3.9 Page

This brick contains all templates required to render the core construct of an HTML page. It will integrate the PBE including the preview device slider and the developer mode icon.

PageGrid

Although rendering for PageGrid and PageGridPlacement is included the intention is to override it in your themes so the actual PageGrid of the your site can be rendered in a suitable way making use of our various other bricks.

Templates

- `Page.ftl` renders the HTML tag.
- `Page._head.ftl` renders the head tag.
- `Page._additionalHead.ftl` renders CSS and JavaScript in head and provides the view hook `VIEW_HOOK_HEAD`.
- `Page._body.ftl` renders the body tag including the PageGrid. It also shows a warning, if JavaScript is disabled.
- `Page.bodyEnd.ftl` renders JavaScript at the end of the body tag and provides the view hook `VIEW_HOOK_END`.
- `PageGrid.ftl` renders the PageGrid and includes the PageGridPlacements.
- `PageGridPlacement.ftl` renders a PageGridPlacement and its items with the default view.
- `CMCSS.asCSSLink.ftl` renders a link tag to include the content of a CMCSS content item.

- `MergeableResources.asCSSLink.ftl` renders a link tag to include the merged CSS.
- `CMJavaScript.asJSLink.ftl` renders a script tag to include the content of a CMJavaScript content item.
- `MergeableResources.asJSLink.ftl` renders a script tag to include the merged JavaScript.

6.3.10 Preview

The `preview` brick enables the fragment preview in *CoreMedia Studio*. When opening a content, the editor will see a preview next to the editing fields on the right side of *Studio*. Install this brick to make sure the preview not only shows the detail view of the content type, but also other predefined views.

Compared to the default preview, the fragment preview displays multiple views of the given content. The different views are rendered as collapsible panels beneath one another. See Figure 6.11, “Example of fragmentPreview Setting Properties” [137] and have a look at an example how to configure which views will be displayed in the fragment preview.

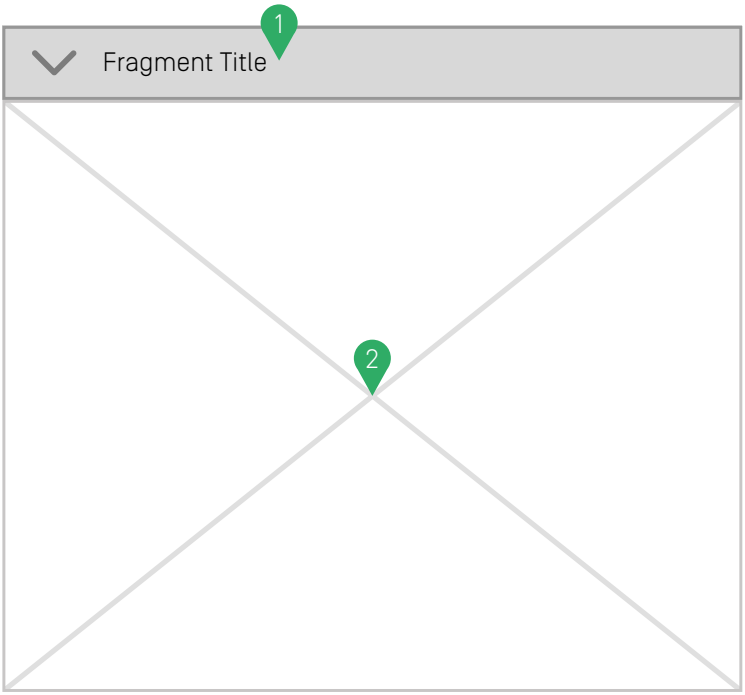

Figure 6.10. Wireframe for preview on desktop

Technical Description

As a frontend developer working with FreeMarker templates, the entry point for any site is `Page.fragmentPreview.ftl`. Per default, this template delegates to `Page.ftl`, unless another template with the same view overrides this behavior. The `Page.fragmentPreview.ftl` in the `preview` brick does exactly this and delegates to `*.asPreview.ftl` templates instead.

Templates and Parameters

- `Page.fragmentPreview.ftl`
- `*.asPreview.ftl`
- `Object.multiViewPreview.ftl`

These `*.asPreview.ftl` templates are used to assign a list of views for the corresponding content type and include the provided `Object.multiViewPreview.ftl` template to render each view in a collapsible panel.

Default views can be configured as follows:

*Assign Default Views in *.asPreview.ftl*

```
<#assign defaultViews=[{
  "viewName": "asTeaser",
  "titleKey": "preview_label_teaser"
 }]/>
```

*Assign Views in *.asPreview.ftl via Content Settings*

Alternatively assign views via `bp.previewTypes` macro, which then returns a list of views configured in Content:

```
<#assign fragViews=bp.previewTypes(cmpage, self, defaultViews)/>

<@cm.include self=self view="multiViewPreview" params={
  "fragmentViews": fragViews
}/>
```

Fragment Preview View Configuration

The `bp.previewTypes` macro retrieves the preview views of an object based on its content type hierarchy or returns the passed default if no views could be found. These preview views can be changed by setting the `fragmentPreview` Struct property in a settings content item, which can either be linked to the `Linked Settings` of the site's root channel or be part of a preview settings json file located in your theme, as recommended. For more information about settings in themes see Section 4.7, “Settings” [49].

The `titleKey` property in the `Linked Settings` and in the example above defines the title of a collapsible panel, displayed in the preview. Since it represents a key, a corresponding entry should be added to a `*.properties` file located in your theme if it does not already exist in the translations included in the brick.

The `viewName` property defines the view type in which the object is rendered. For example `asHeroBanner`. To use the default simply put `DEFAULT`.

With help of the `viewParams` property, parameters can be send to the template for further configuration.

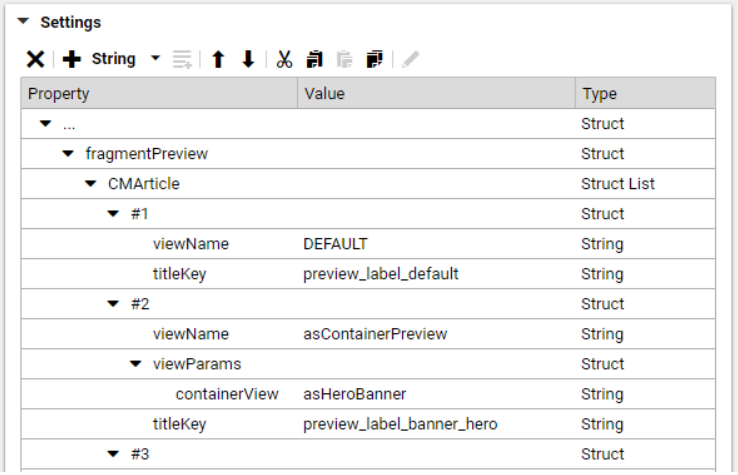

Figure 6.11. Example of fragmentPreview Setting Properties

6.3.11 Slick Carousel

The `slick-carousel` brick provides templates, styling, and functionality for displaying content in a carousel based on slick.

Technical Description

This brick uses the library slick, especially the fork slick-carousel-no-font-no-png. Please check the official documentation about features and configuration.

The `slick-carousel` provides an API to create custom carousels which can be used in themes or other bricks.

NOTE

Please note that this brick contains `JavaScript` and `SASS` files, what will automatically be installed, if you add the brick to your theme `package.json`. See Section 5.3, "Using Bricks" [67] to learn how to install a brick in your theme.

API

The brick provides the FreeMarker Library via `src/freemarkerLibs/slickCarousel.ftl`. Please check the template for further information.

You can define a custom prefix for the rendered carousels via the `$cm-slick-carousel-prefix` and decide if custom arrow styles should be enabled via `$cm-slick-carousel-custom-arrows-enabled`.

6.3.12 Utilities

This brick contains different utilities for SASS, templates and JavaScript that provide reusable and helpful macros and functions to use in bricks and themes.

JavaScript Utilities

For JavaScript the brick offers functions like our logger, to extend jQuery and others. They are all documented in their source files and to use them they need to be imported in the code first like in the following example:

```
import { log } from "@coremedia/brick-utils";

log("Logging something");
```

Example 6.4. Example import of the logger

Sass Utilities

The Sass mixins and functions are available in a theme or brick without explicit import and can be used like the following example:

```
.button {
  @include center-absolute();
}
```

Example 6.5. Example use of center-absolute mixin

FreeMarker Utilities

The FreeMarker macros and functions need to be imported in the templates where they are to be used. For example:

```
<#import
"*/node_modules/@coremedia/brick-utils/src/freemarkerLibs/components.ftl" as
 components />
```

```
<@components.button text=cm.getMessage("button_text") attr={"type": "submit"}
 />
```

Example 6.6. Example use of the button macro

6.4 Example Bricks

In contrast to Section 6.3, “Bricks” [121] the bricks of this category are only for demonstration purposes of different features that can be build with the Frontend Workspace. None of these bricks is meant to be stable across different CoreMedia versions. While CoreMedia will mention changes like new features and major adjustments in the release notes there will be no direct upgrade path for example bricks.

Just like Section 6.1, “Example Themes” [103] can be found in the `themes/` folder, all example bricks can be found in the `bricks/` folder of the frontend workspace. Every package is contained in a single directory prefixed with `example-`.

CAUTION

The theme build will trigger a warning if you are using an example brick in your own themes. In case you want to reuse an example brick check the chapter: Section 5.4, “Using an Example Brick” [69].

Available Example Bricks

- Section 6.4.1, “Example 360-Spinner” [141]
- Section 6.4.2, “Example Carousel Banner” [142]
- Section 6.4.3, “Example Cart” [144]
- Section 6.4.4, “Example Detail” [145]
- Section 6.4.5, “Example Download-Portal” [147]
- Section 6.4.6, “Example Elastic Social” [147]
- Section 6.4.7, “Example Footer” [147]
- Section 6.4.8, “Example Fragment-Scenario” [150]
- Section 6.4.9, “Example Hero Banner” [150]
- Section 6.4.10, “Example Landscape Banner” [153]
- Section 6.4.11, “Example Left Right Banner” [155]
- Section 6.4.12, “Example Navigation” [158]
- Section 6.4.13, “Example Popup” [162]
- Section 6.4.14, “Example Portrait Banner” [163]
- Section 6.4.15, “Example Product Assets” [166]
- Section 6.4.16, “Example Search” [167]
- Section 6.4.17, “Example Shoppable-Video” [171]
- Section 6.4.18, “Example Square Banner” [173]
- Section 6.4.19, “Example Tag-Management” [174]

6.4.1 Example 360-Spinner

This brick provides the 360 Spinner functionality, to render the content type 360°-View in your theme. It displays a set of images that you can rotate to have a view around a product and therefore includes templates, SCSS and JavaScript.

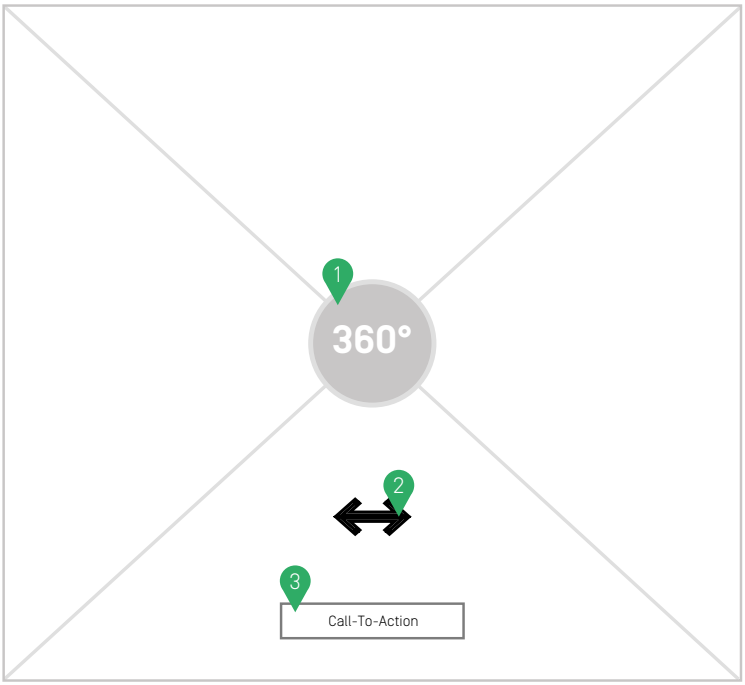

Figure 6.12. Wireframe of 360°-Spinner on desktop

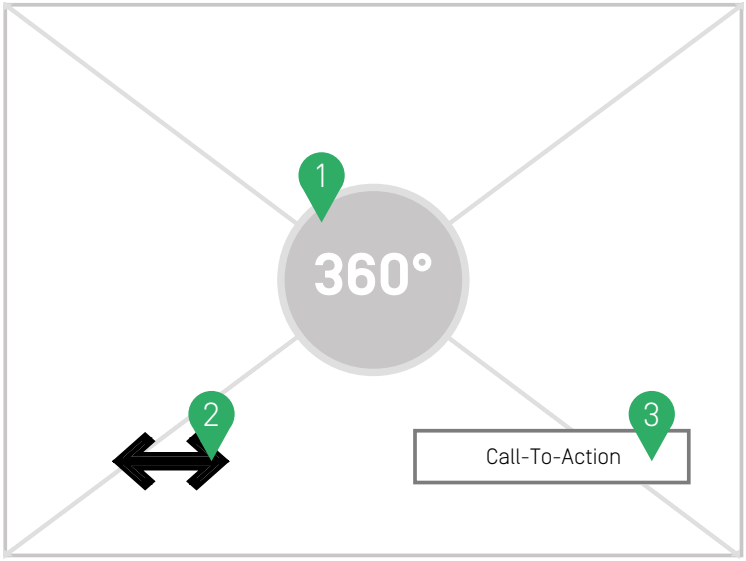

Figure 6.13. Wireframe of 360°-Spinner on mobile

1. Adds the spinner-icon to the brick, on click the brick starts to load in the images for the preview
2. On mouseover of the spinner-brick, the cursor changes into the double arrows to indicate interactivity with the brick
3. CTA is only available as a hero element and replaces the spinner icon

6.4.2 Example Carousel Banner

The brick provides templates and CSS styles for displaying many content types and commerce objects as a carousel banner. The example is based on the API brick Section 6.3.1, “Default-Teaser” [121] and Section 6.3.11, “Slick Carousel” [137].

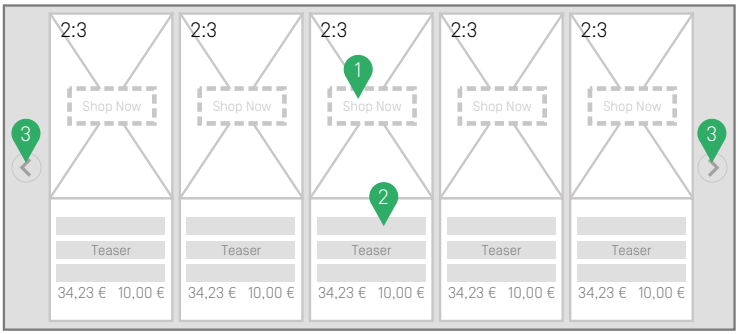

Figure 6.14. Wireframe for carousel-banner on desktop

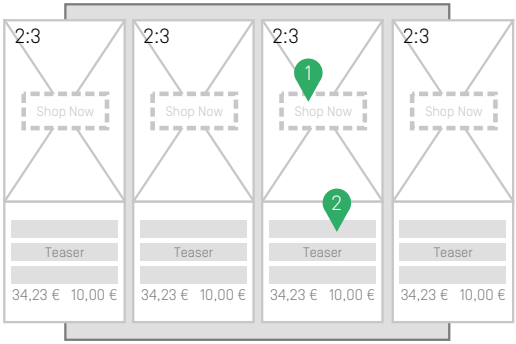

Figure 6.15. Wireframe for carousel-banner on mobile

1. The shop-now button feature is utilized.

2. Teaser title and teaser text are shown. For products the list price and (if existing) the offer price is shown.
3. If necessary the items will be displayed as a carousel with arrows. For mobile devices there is touch support to control which item is shown.

The banner supports a corresponding view type "carousel" which can be used for placements and collections. If more than one item is in the same container as carousel, they will be displayed as a carousel showing multiple items at once based on the available screen size. It has arrows and touch support to control which items are shown. The visual output of the carousel is decided by the content assigned to it, for example, teaser, image-maps, video, product, category...

Using the Brick

As shown in the example below, a carousel banner can be displayed by including the corresponding content type with the `asCarouselBanner` view.

```
<@cm.include self=self view="asCarouselBanner"/>
```

The same view is defined for containers (for example, a CMCollection or Placement) to render multiple items inside a carousel.

Video Behavior

The carousel banner will render a picture linked to the banner item or an empty placeholder and play the linked video in a popup (only if the popup brick is enabled). The video will automatically start to play as soon as the popup is opened. You can hide the controls and mute or loop the video by setting the corresponding video options in CoreMedia Studio.

6.4.3 Example Cart

The brick provides templates, CSS styles and JavaScript for to handle a cart.

Using the Brick

Rendering a cart

First of all you need to have a cart. The most simple way to achieve this is by using `cm.substitute`.

```
<#assign cart=cm.substitute("cart") />
```

You can also utilize a CMAction and use `cart` as its id.

After retrieving the cart it can be rendered using the "asCart" view:

```
<@cm.include self=cart view="asCart"/>
```

Add-To-Cart Button

An add-to-cart button can be added via the provided FreeMarker library `cart.ftl`. You need to provide a `com.coremedia.livecontext.ecommerce.catalog.Product` as the macro needs some information from this bean.

```
<#-- @ftlvariable name="self"
type="com.coremedia.livecontext.ecommerce.catalog.Product" -->

<#import
"*/node_modules/@coremedia-examples/brick-cart/src/freemarkerLibs/cart.ftl"
 as cart />

<@cart.addToCartButton product=self.product!cm.UNDEFINED
                       enableShopNow=true />
```

Please check the FreeMarker library for information about the different parameters.

6.4.4 Example Detail

The detail brick renders documents in a full page layout. This view is the most detailed, containing the title and text, media elements, a list of authors and related content. The detail brick provides templates, JavaScript, localizations and CSS styles for detail views. Templates exist for all kinds of `CMTeasable` and as special variants for certain other types, such as Products, Persons, Videos etc.

Detail View

As shown in the example below, a detail view can be displayed by including the corresponding content type with the `detail` view.

```
<@cm.include self=self view="detail"/>
```

The detail view template works with all types and subtypes of type `com.coremedia.blueprint.common.contentbeans.CMTeasable`. The following special views exist:

- `CMAudio.detail.ftl`
- `CMGallery.detail.ftl`
- `CMPerson.detail.ftl`
- `CMProduct.detail.ftl`
- `CMVideo.detail.ftl`

Using the Brick

As shown in the example below, a full page layout can be displayed by including the corresponding content type with the `detail` view. You can also pass additional CSS classes as parameters to apply custom styling to your detail view.

```
<@cm.include self=self view="detail"/>
```

The detail view template works with all types and subtypes of type `com.coremedia.blueprint.common.contentbeans.CMTeasable`. The following special views exist:

- `CMAudio.detail.ftl`
- `CMGallery.detail.ftl`
- `CMPerson.detail.ftl`
- `CMProduct.detail.ftl`
- `CMVideo.detail.ftl`
- `CMImageMap.detail.ftl` (part of the ImageMap brick)

To configure the behavior of the template you can add the following parameters to the `cm.include` tag:

Parameter	Type	Default	Description
`blockClass`	String	`"cm-details"`	A base name that will be used for CSS classes attached to the elements rendered by the template.
`renderAuthors`	Boolean	`true`	Whether to display the author of the document or not.
`renderDate`	Boolean	`true`	Whether to display the date or not.

Parameter	Type	Default	Description
`renderRelated`	Boolean	`true`	Whether to display the related content or not.
`renderTags`	Boolean	`true`	Whether to display a list of tags or not.
`relatedView`	String	`"asRelated"`	The name of the view to render related content in.

Table 6.13. Parameters of the Detail View

Video Behavior

Videos in the detail view will be displayed inline. You can hide the controls, mute and loop the video or enable autoplay by setting the corresponding video options in CoreMedia Studio. The detail view will not display additional preview pictures linked to the video.

6.4.5 Example Download-Portal

The Download-Portal offers an informative and versatile UI and functionality for downloading assets. It provides templates, SCSS and JavaScript.

6.4.6 Example Elastic Social

This brick acts as an entry point into CoreMedia Elastic Social. You should include this brick, if you want to use user management, reviews or ratings on your site.

Using the Brick

By loading the brick, existing templates are copied from the Elastic Social extension.

6.4.7 Example Footer

This brick renders a simple footer with two placements - `footer` and `footer-navigation`. The Footer placement displays a list of `CMTeasable` next to a copyright information and social media icons. The Footer Navigation placement displays

an additional navigation above the actual footer and can handle `CMTeasable` as content.

While `CMTeasable` are displayed as a simple link in the footer navigation, there are additional templates for `CMSitemap`, `Navigation` and `CMCollection` to display their elements as list entries. Custom HTML can displayed in these lists by using `CMHTML`.

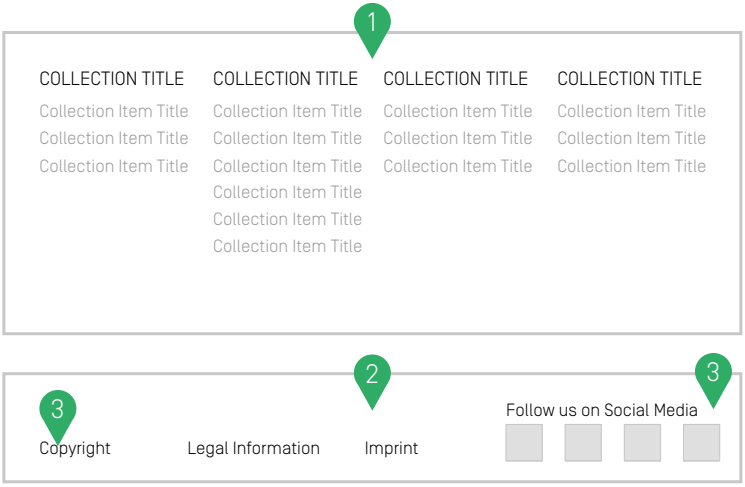

Figure 6.16. Wireframe of footer on desktop

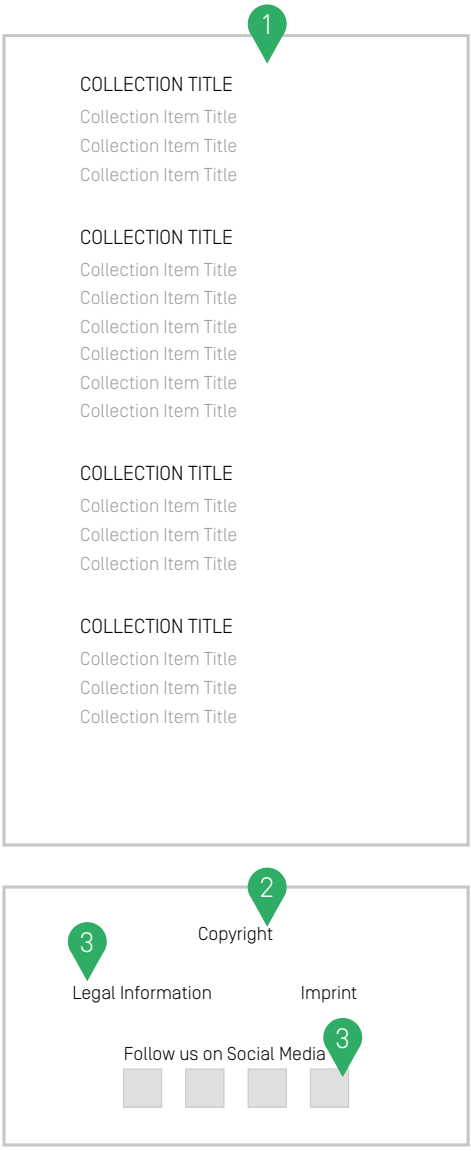

Figure 6.17. Wireframe of footer on mobile

1. Top section: linkable content dependent footer navigation

2. Bottom section: legal information is rendered as another link list
3. Copyright and social media are set in the brick's code and cannot be edited in Studio

6.4.8 Example Fragment-Scenario

This brick adds support for rendering external requested fragments. The typical use case for this brick is the commerce-led or hybrid scenario, where CoreMedia delivers fragments for an eCommerce system.

6.4.9 Example Hero Banner

The brick provides templates and CSS styles for displaying many content types and commerce objects as a hero banner. The example is based on the API brick Section 6.3.1, “Default-Teaser” [121] and Section 6.3.11, “Slick Carousel” [137].

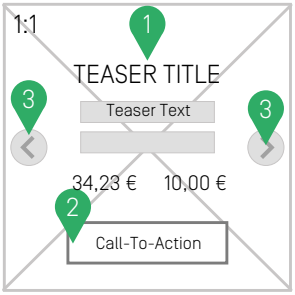

Figure 6.18. Wireframe for hero-banner on desktop

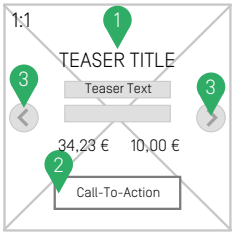

Figure 6.19. Wireframe for hero-banner on mobile

1. Teaser title and teaser text are shown. For products the list price and (if existing) the offer price is shown.
2. The call-to-action button feature is utilized and also placed on top of the picture.
3. If more than one item is in the same container as hero, they will be displayed as a carousel with arrows. For mobile devices there is touch support to control which item is shown.

The banner supports a corresponding view type "hero" which can be used for placements and collections. The visual output of the hero is decided by the content assigned to it, for example, teaser, image-maps, video, product, category...

Using the Brick

As shown in the example below, a hero banner can be displayed by including the corresponding content type with the `asHeroBanner` view.

```
<@cm.include self=self view="asHeroBanner"/>
```

The same view is defined for containers (for example, a CMCollection or Placement) to render multiple items inside a hero carousel.

Video Behavior

Videos in hero banners will be displayed inline. The videos will always be autoplayed, muted, looped and displayed with hidden controls. These settings can not be overwritten in the video options in CoreMedia Studio. Hero banners will not display additional preview pictures linked to the video.

6.4.10 Example Landscape Banner

The landscape-banner brick provides templates and CSS styles for displaying many content types and commerce objects as a landscape banner. The example is based on the API brick Section 6.3.1, "Default-Teaser" [121].

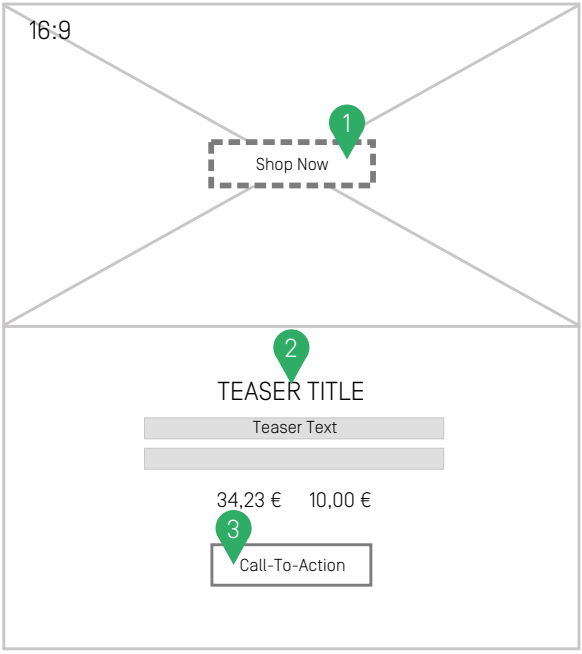

Figure 6.20. Wireframe for landscape-banner

1. The shop-now button feature is utilized.
2. Teaser title and teaser text are shown. For products the list price and (if existing) the offer price is shown.
3. The call-to-action button feature is utilized.

Additional information like title, text and Call-to-Action buttons are placed below the picture.

The banner supports a corresponding view type "landscape" which can be used for placements and collections.

Using the Brick

As shown in the example below, a landscape banner can be displayed by including the corresponding content type with the `asLandscapeBanner` view.

```
<@cm.include self=self view="asLandscapeBanner"/>
```

The same view is defined for containers (for example, a CMCollection or Placement) to render multiple items inside a grid containing multiple landscape banners per row based on the available screen size

Video Behavior

The landscape banner will render a picture linked to the banner item or an empty placeholder and play the linked video in a popup (only if the popup brick is enabled). The video will automatically start to play as soon as the popup is opened. You can hide the controls and mute or loop the video by setting the corresponding video options in CoreMedia Studio.

6.4.11 Example Left Right Banner

The left-right-banner brick provides templates and CSS styles for displaying many content types and commerce objects as a left-right banner. The example is based on the API brick Section 6.3.1, “Default-Teaser” [121].

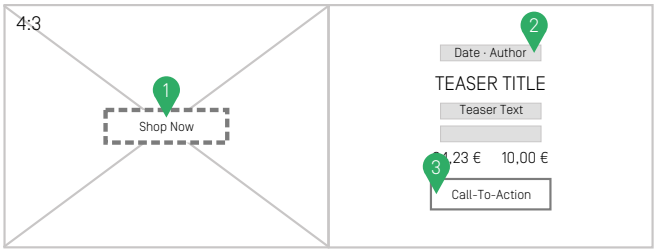

Figure 6.21. Wireframe for left-right-banner

1. The shop-now button feature is utilized.
2. Teaser title and teaser text are shown. For products the list price and (if existing) the offer price is shown.
3. The call-to-action button feature is utilized.

When used in a container with multiple items the left and right half of the banner alternate.

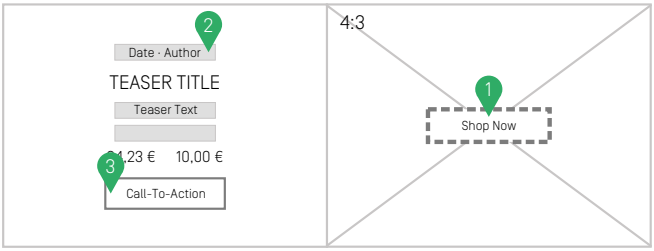

Figure 6.22. Wireframe for left-right-banner (alternative)

The banner supports a corresponding view type "left-right" which can be used for placements and collections.

Using the Brick

As shown in the example below, a left-right banner can be displayed by including the corresponding content type with the `asLeftRightBanner` view.

```
<@cm.include self=self view="asLeftRightBanner"/>
```

The same view is defined for containers (for example, a CMCollection or Placement) to render multiple items among themselves.

Video Behavior

Videos in left-right banners will be displayed inline. You can hide the controls, mute the video or enable autoplay by setting the corresponding video options in CoreMedia Studio. Please note that the autoplay setting will also affect the loop and controls configuration. Loop is enabled for autoplayed videos and disabled otherwise. In addition to that, the video controls will automatically be hidden if autoplay is enabled, no matter the hide

controls configuration. Left-right banners will not display additional preview pictures linked to the video.

6.4.12 Example Navigation

The navigation brick provides a navigation that allows the user to browse through the site. It is capable of rendering links to content pages, commerce categories or any other suitable `CMTeasable` implementations.

NOTE

Most subtypes of `CMCollection` are supported but they will be rendered particularly. If a collection does not have a teaser title or if it returns only one content then it is handled transparently. The navigation then shows the containing content at the level of the collection instead of a level below.

The navigation displays a configurable number of navigation levels and will be rendered as an overlay menu or as an additional menu below your site's header toolbar. The default depth of the navigation is set to 3 levels. If you want to have additional levels you might need to add appropriate styling as the example only contains styling for the default depth.

Figure 6.23. Wireframe for navigation on desktop

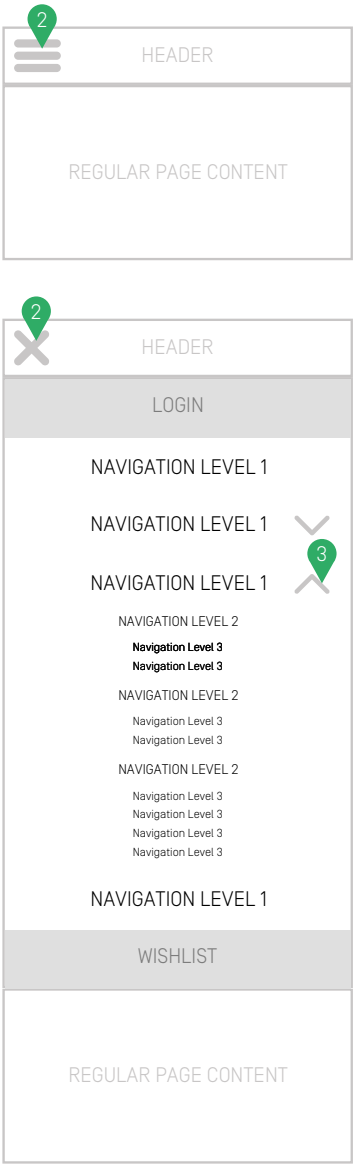

Figure 6.24. Wireframe for navigation on mobile

1. The Navigation title inherits its name and link destination from its active navigation level 1
2. On mobile: the navigation can be accessed through the hamburger menu.
3. On mobile: if the navigation contains more than one level, a caret icon appears next to the entry
4. Picture can be set in Studio

Technical Description

The navigation can be displayed by including the `Page` with the `navigation` view.

```
<header>
  ...
  <ul>
    ...
    <@cm.include self=cmpage view="navigation" params={
      "cssClass": "custom-styled-navigation"
    }/>
    ...
  </ul>
  ...
</header>
```

You can also use the following settings as parameters in your include to apply additional styles and adjust the behavior of the navigation:

- `Page.navigation.ftl`

Parameter	Type	Default	Description
`cssClass`	`String`	`""`	An additional CSS class that will be added to the Navigation.
`childrenCssClass`	`String`	`""`	An additional CSS class that will be added to the children of the Navigation.
`showPicturesInNavigation`	`Boolean`	`true`	Set to false to hide pictures of `CMTeasables` and Catalog Categories in the Navigation.

Table 6.14. Parameters of the Navigation

To make the navigation appear when a certain header placement is loaded, you can also include the `PageGridPlacement.asNavigationHeader.ftl` example, which is part of the brick. A closer look inside this template can also provide insight on how to use the navigation in your own templates. The following example shows an excerpt of a `PageGridPlacement.ftl`, which includes the navigation:

```
<#if self.name! == "header">
  <@cm.include self=self view="asNavigationHeader"/>
<#else>
  ...
</#if>
```

The maximum depth of the navigation can be changed via setting `navigation_depth`.

Known Limitations

There are basically no limitations in terms of how an editor can build a navigation in the repository. The navigation brick cannot cover all these cases. The following list describes the most obvious limitations:

- When content appears multiple times in the navigation and it is selected by the website user, all occurrences are highlighted as active.
- Active items cannot be properly highlighted when nesting collections and pages. For example, when linking from a collection to a page which is already part of the navigation, most likely not all levels will be highlighted as active up to the currently selected page.

6.4.13 Example Popup

The Popup brick includes the Magnific Popup jQuery plugin. It extends templates of other bricks and renders overlays for certain content types, such as Image Maps, Videos or eCommerce Products. It can also easily be used to open text and images in a popup or in a full screen gallery. Therefore, it includes templates, SCSS and JavaScript.

Extending the Image Map

The `CMImageMap._areasMap.ftl` of the Image Map Brick will be overwritten with the including one and a click on a hot zone will open the linked content in a popup gallery. Arrows on both sides will slide through the contents of all visible hot zones. On mobile devices the popup is fullscreen.

Using the Video Popup

Initialize magnific popup for video popup.

```
<a href="..." data-cm-popup="${videoLink}"> ...</a>
```

CoreMedia will automatically find and initialize a video popup opener for any element that contains this data attribute.

Using the Popup for Shop Now

The template `LiveContextProductTeasable._shopNow.ftl` adds the popup to product teaser, if the shop now functionality is enabled. This overwrites the template of the default-teaser brick.

Using Magnific Popup for other use cases

The popup functionality can be used in every view and works out of the box. Just add the data-attribute `mfp-src` to any element with the id of the DOM element, which should be displayed in the popup. For more information check the official documentation of Magnific Popup.

Additional Resources

- `Popup_de.properties`
- `Popup_en.properties`

Dependencies

This brick has dependencies on the npm packages `jQuery` and `magnific-popup`, some `Freemarker` and `JavaScript` from the frontend `lib` folder.

- Media Brick
- Image Maps Brick
- Default Teaser Brick
- MediaElement Brick

6.4.14 Example Portrait Banner

The portrait-banner brick provides templates and CSS styles for displaying many content types and commerce objects as a portrait banner. The example is based on the API brick Section 6.3.1, “Default-Teaser” [121].

Figure 6.25. Wireframe for portrait-banner on desktop

Figure 6.26. Wireframe for portrait-banner on mobile

1. The shop-now button feature is utilized.
2. Teaser title and teaser text are shown. For products the list price and (if existing) the offer price is shown.
3. The call-to-action button feature is utilized.

The picture associated with a portrait banner fills the upper area. Additional information like title, text and Call-to-Action buttons are placed below the picture.

The banner supports a corresponding view type "portrait" which can be used for placements and collections.

Using the Brick

As shown in the example below, a portrait banner can be displayed by including the corresponding content type with the `asPortraitBanner` view.

```
<@cm.include self=self view="asPortraitBanner"/>
```

The same view is defined for containers (for example, a CMCollection or Placement) to render multiple items inside a grid containing multiple portrait banners per row based on the available screen size

Video Behavior

The portrait banner will render a picture linked to the banner item or an empty placeholder and play the linked video in a popup (only if the popup brick is enabled). The video will automatically start to play as soon as the popup is opened. You can hide the controls and mute or loop the video by setting the corresponding video options in CoreMedia Studio.

6.4.15 Example Product Assets

This brick provides templates, SCSS and JavaScript to render content from the CoreMedia CMS as a fragment on a product detail page for an augmented product.

It will utilize the assigned catalog items of Picture, Video and 360° View content items to create a slideshow which can be controlled by an underlying carousel.

When hovering over a picture, a zoom window appears on the right side of the slideshow taking the available space of the surrounding container. Per default the container is determined by finding the closest parent matching the DOM selector `.row`. You can change the selector in the `productAssets` settings by overriding the entry `zoom.containerSelector`.

```
{
  "productAssets": {
    "zoom": {
      "containerSelector": ".my-special-class"
    }
  }
}
```

Videos do not have a zoom window but they can be played by clicking the rendered play button which will open a popup window.

When assigning a 360° View to a product it can be rotated after it has been selected in the carousel.

6.4.16 Example Search

The search brick provides templates, SCSS and translations to render a search input field, a search results page with a configurable amount of results and a filter panel. To get additional entries there is a "Load More" button beneath the list and a spinner is shown while loading. The search results can be listed sorted by date or by relevance.

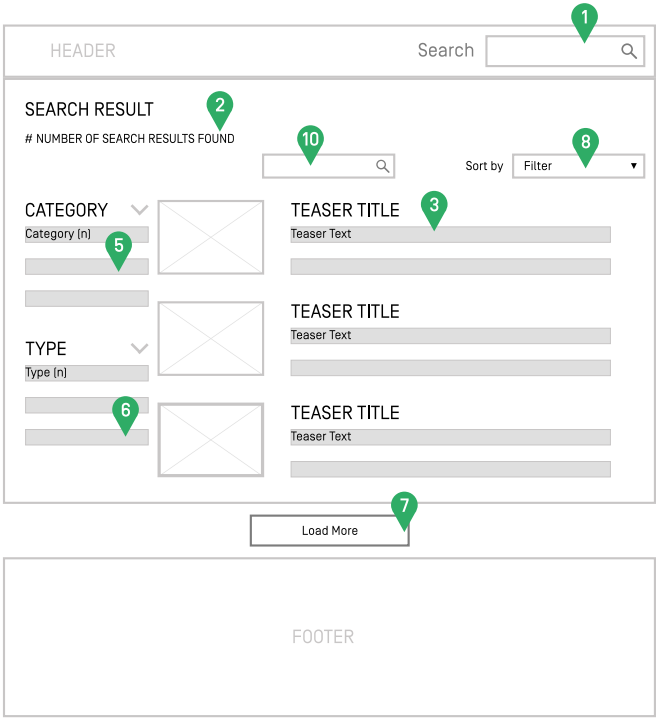

Figure 6.27. Wireframe of search on desktop

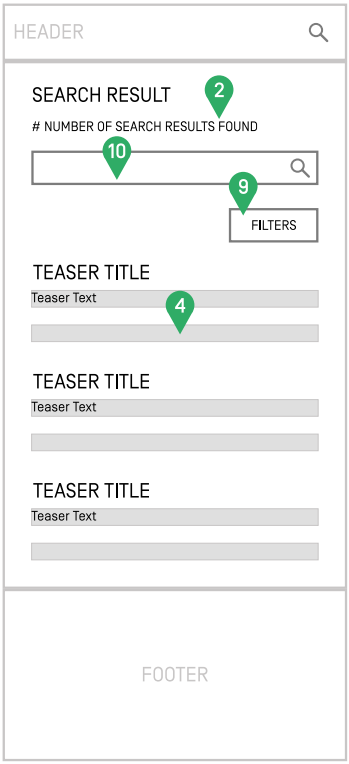

Figure 6.28. Wireframe of search on mobile

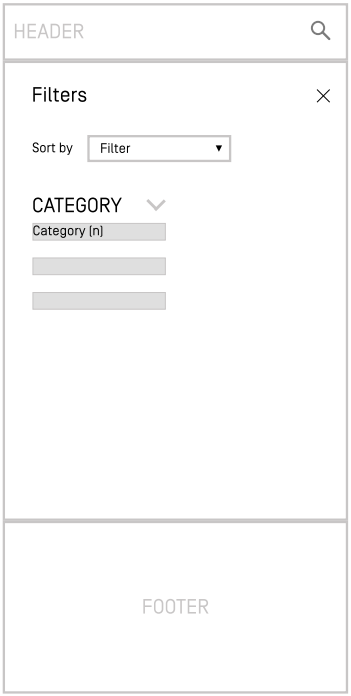

Figure 6.29. Wireframe of search on mobile with open filter menu

1. The brick contains templates to render a search field into a page
2. While showing search results, the brick displays the found number of entriess
3. Example of a search result: with date (first position / different typo), title, text and picture
4. On mobile: images are not shown by default
5. Collapsable category list (n) = number of search results in category
6. Collapsable type list (n) = number of search results of the same type
7. Load more search results (button)
8. Filter dropdown (Relevance and Date)
9. On mobile the filter dropdown is a button that leads to the filter menu

Technical description

The `search` brick works out of the box in any theme by adding the dependency. Add a search configuration to your site as described in the next section. The search works as a Single Page Application. All filters and links reload the results via AJAX.

Templates

- `SearchActionState.asSearchResultPage.ftl` renders a Search Result Page including a title, number of results.
- `SearchActionState.asSearchField.ftl` renders a search field with label, input field and submit button.
- `SearchActionState.asResultList.ftl` renders the results as list.
- `CMTeasable.asSearchResult.ftl` renders single search result including title, picture and text to the list.

> **NOTE**
>
> Please note that this brick contains `JavaScript` and `SASS` files that are automatically installed if you add the brick to your theme `package.json`. See Section 5.3, "Using Bricks" [67] to learn how to install a brick in your theme.

Configuration

In order to use the search brick, there must be a Setting called `searchAction` linking to an existing `CMAction` content item. For the search result page add a Setting called `searchChannel` linking to an existing `Page` content item. This should also include the searchAction in the PageGrid to render search results.

For a description of the search functions visit Section 5.4.21, "Website Search" in *Blueprint Developer Manual* or go to ???? for the detailed API guide of the `Search Configuration` settings.

Including in templates

```
<#assign searchAction=bp.setting(self,"searchAction", {})/>
<@cm.include self=searchAction view="asSearchField" />
```

Example 6.7. Example template to render the search form

Using a placeholder in content

Add a `CMPlaceholder` with a layout variant `search` into the site for rendering a simple search field where it is required.

6.4.17 Example Shoppable-Video

This brick provides templates, SCSS and JavaScript to use shoppable videos on a website. It allows you to display products next to a video at a predefined time.

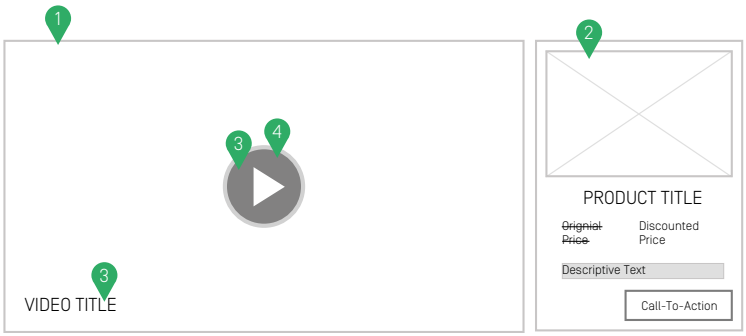

Figure 6.30. Wireframe of shoppable video

1. Can only be rendered as a teaser
2. Renders a preview of a product, for example from the video at a specified time
3. Disappears when the video starts playback
4. The brick plays the video inline and not in a pop-up

Technical Description

The shoppable video can be explicitly configured in Studio to show product teasers at certain times in a video. Those product teasers will then be rendered right next to the video one at a time. An additional teaser image can be shown when the video is loaded and until started via click on the play icon.

Templates and Parameters

In order to use a shoppable video, you can either include it as a teaser or a hero teaser by using the following templates.

- `CMVideo.hero[shoppable].ftl`
- `CMVideo.teaser[shoppable].ftl`

The templates can be included in your theme as follows:

```
<@cm.include self=self view="hero" params={"blockClass": "example-class"}/>
```

To configure the behavior of the template you can add the following parameters to the `cm.include` tag:

Parameter	Type	Default	Description
`additionalClass`	`String`	`""`	A string as CSS class added to the shoppable video container.
`blockClass`	`String`	`""`	This will add a CSS class to elements of the image map, all beginning with the provided string.
`renderDate`	`Boolean`	`true`	If the image map should show a date.
`renderTeaserText`	`Boolean`	`false`	Enables rendering the teaser text in addition to the shoppable video.
`timelineEntries`	`Array`	`[]`	An array of product teasers and the time points at when to be shown.
`overlay`	`Object`	`{}`	An object with overlay settings for the product teaser. All as Boolean and defaulting to true: `displayTitle, displayShortText, displayPicture, displayDefaultPrice, displayDiscountedPrice, displayOutOfStockLink`

Table 6.15. Parameters of the Image Map

Dependencies

Please note that the `shoppable-video` brick has dependencies on *jQuery* and the *Media* brick for responsive images, for example, as a teaser image, shown before the video starts. Also, the *mediaelement* brick to provide the media element API for the video to have access to the exact timing of the video so it can display product teasers in the specified moments using the teaser macro by the *default-teaser* brick.

- jQuery

- Media Brick
- MediaElement Brick
- Default Teaser Brick

6.4.18 Example Square Banner

The square-banner brick provides templates and CSS styles for displaying many content types and commerce objects as a square banner. The example is based on the API brick Section 6.3.1, “Default-Teaser” [121].

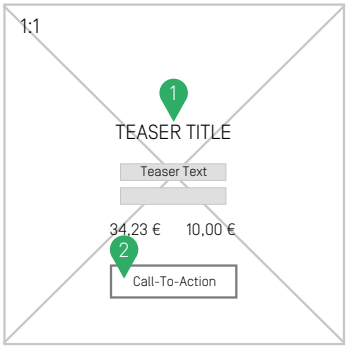

Figure 6.31. Wireframe for square-banner

1. Teaser title and teaser text are shown on top of the picture.
2. The call-to-action button feature is utilized and also placed on top of the picture.

The banner supports a corresponding view type "square" which can be used for placements and collections.

Using the Brick

As shown in the example below, a square banner can be displayed by including the corresponding content type with the `asSquareBanner` view.

```
<@cm.include self=self view="asSquareBanner"/>
```

The same view is defined for containers (for example, a CMCollection or Placement) to render multiple items inside a grid containing multiple square banners per row based on the available screen size.

Video Behavior

Videos in square banners will be displayed inline. You can hide the controls, mute the video or enable autoplay by setting the corresponding video options in CoreMedia Studio. Please note that the autoplay setting will also affect the loop and controls configuration. Loop is enabled for autoplayed videos and disabled otherwise. In addition to that, the video controls will automatically be hidden if autoplay is enabled, no matter the hide controls configuration. Square banners will not display additional preview pictures linked to the video.

6.4.19 Example Tag-Management

The `brick-tag-management` adds support for Tag Management Systems to the theme. It overrides the following three templates from `brick-page`:

- `Page._additionalHead.ftl`
- `Page._body.ftl`
- `Page._bodyEnd.ftl`

The Tag Management System snippets can by configured during runtime by technical editors. See Section 5.4.25, “Tag Management” in *Blueprint Developer Manual* for more details on configuration options.

> **NOTE**
>
> This brick does not completely support the fragment scenario with a commerce system yet.

6.5 CoreMedia FreeMarker Facade API

The *CAE* web application and tag libraries are based on the latest *FreeMarker* 2.3.x syntax. For more information see Section 4.3.4, "Writing Templates" in *Content Application Developer Manual* and FreeMarker documentation.

The taglibs `cm` and `preview` are implicitly available in any FreeMarker template view rendered by the *CAE* and are needed for main functionality. Other taglibs, like `bp`, are part of *CoreMedia Blueprint* and offer additional and helpful functions depending on the extension and context they are part of.

Auto-Import of FreeMarker Functions and Macros

In order to create your own Taglib please take a look at Section 4.3.4.2, "Advanced Patterns for FreeMarker Templates" in *Content Application Developer Manual*. You need to add the corresponding FreeMarker file to the `freemarkerConfigurer` bean's property `autoImports` in the according Spring configuration, for example like in `blueprint-freemarker-views.xml`.

Available APIs

6.5.1 CoreMedia (cm)

The CoreMedia FreeMarker API provides helpful macros and functions and is implicitly available in any FreeMarker template view rendered by the CAE. It uses the namespace `cm` for template calls.

UNDEFINED

`cm.UNDEFINED`

Returns a value representing that something is undefined as the FreeMarker template language has no build in support for `null` or `undefined` values. You will most likely encounter this value as a return value of various functions provided by our FreeMarker API. The value can be interpreted as a Boolean (`false`), a string (`""`), a sequence

[[ ]] or a hash [{ }] including all build-ins without needing additional checks to prevent rendering errors.

For example: Using the build-in `?has_content` the code `cm.UNDEFINED?has_content` would return `false` which is exactly what would be expected from an empty string.

Use this value as a default value for parameters that should be ignored if not defined, like so:

```
<@cm.include self=self params={ "param1": param1!cm.UNDEFINED }/>
```

It also tells an include not to fail if the parameter for "self.related" is undefined:

```
<@cm.include self=self.related!cm.UNDEFINED/>
```


CAUTION

You might need this to distinguish `cm.UNDEFINED` value from an empty string or similar for various reasons. Please note that you cannot use build-ins such as `==` or `!=` to check if a given value is `cm.UNDEFINED` as its value is equal to `false` and an empty string [`""`].

Please use one of functions described in the following sections instead.

`cm.isUndefined(value)`

Returns `true` if the given value is `cm.UNDEFINED` otherwise `false`.

`cm.notUndefined(value, fallback)`

Returns the value if it is not `cm.UNDEFINED` otherwise it will return the given fallback.

```
<#assign valueToUse=cm.notUndefined(providedValue, "hello") />
```

Example 6.8. Making sure that a provided value is not cm.UNDEFINED

`cm.include`

This macro is the most important one. It includes a template for an object (self), using the view dispatcher instead of FreeMarker's built in include function. With the view parameter you can determine a specific template. Requires a template/view to be defined

for such an object. For more information see Section 4.3.4, "Writing Templates" in *Content Application Developer Manual*.

Parameter	Required	Description
self	✔	The target object for the view.
view		A specific template for this object.
params		Pass parameters into the included template.

Table 6.16. Parameters of cm.include

In this example the template `CMArticle.teaser.ftl` would be included without rendering a button, assuming that "article" has the type `CMArticle`.

```
<@cm.include self=article view="teaser" params={"showButton": false}/>
```

Example 6.9. Include a template with view and parameters.

`cm.getLink(target, [view], [params])`

Create a link to the object passed as "target" in the given view and return the URL as a string. Requires a link scheme to be defined for the target object. If the target object is cm.UNDEFINED, an empty string is returned. For more information see paragraph "Linking" in Section 4.3.4, "Writing Templates" in *Content Application Developer Manual*.

Parameter	Required	Description
target	✔	Object of which to render the link to.
view		String to specify a special view.
params		additional parameters given as a map.

Table 6.17. Parameters of cm.getLink

```
<a href="${cm.getLink(self)}">linktext</a>
```

Example 6.10. Returns the URL to this page.

`cm.getIntegrityHash(target)`

Create a Subresource Integrity (SRI) hash to the object passed as "target" and return the base64-encoded sha512 string. It's defined for `<link>` and `<script>` elements and it is used by CSS and JS templates in the blueprint by default.

Parameter	Required	Description
target	✔	Object of which to create the integrity hash for.

Table 6.18. Parameters of cm.getIntegrityHash

```
<#assign integrityHash="${cm.getIntegrityHash(self)}"/>
<#assign integrity=integrityHash?has_content?then('
integrity="${integrityHash}"', "") />
<link href="${cssLink}"${integrity?no_esc}>
```

Example 6.11. Renders the hash for a given CSS content.

`cm.dataAttribute`

Renders a serialized data attribute for HTML elements.

Parameter	Required	Description
name	✔	Name for the attribute (the "data-" prefix is not added automatically)
data	✔	An object containing values.

Table 6.19. Parameters of cm.dataAttribute

`cm.hook`

Renders the results of all `com.coremedia.objectserver.view.events.ViewHookEventListener` implementations that match the given type of self and that support the given ID and the parameters. For more information see Section 4.3.3.9, “View Hooks” in *Content Application Developer Manual*.

Parameter	Required	Description
id	✔	String as identifier for the ViewHookEvent.

Parameter	Required	Description
self		The object that the corresponding listeners have to support. Optional but defaults to "self" object from template context.
params		The parameters passed to the listener through the FreeMarker macro.

Table 6.20. Parameters of cm.hook

```
<@cm.hook id="page_end"/>
```

Example 6.12. Setting a template hook with id "page_end".

`cm.getId(self)`

Determine this object's id through the IdProvider and return the id as a string.

Parameter	Required	Description
self	✔	Object to get ID of.

Table 6.21. Parameters of `cm.getId`

Header

`cm.responseHeader`

Sets an HTTP response header. If the response is already committed, the macro will fail. For more information see Section 4.3.4, "Writing Templates" in *Content Application Developer Manual*.

Parameter	Required	Description
name	✔	Name of the response header as String.
value	✔	Value for the response header as String.

Table 6.22. Parameters of cm.responseHeader

```
<@cm.responseHeader name="Content-Type" value="text/html; charset=UTF-8"/>
```

Example 6.13. Set the content type for the HTTP response header.

`cm.getRequestHeader(name)`

Get an HTTP request header.

Parameter	Required	Description
`name`	✔	Name of the header that should be returned.

Table 6.23. Parameters of cm.getRequestHeader

Parameter

`cm.localParameter(key, [defaultValue])`

Returns a parameter from the `localParameters` map by given name or falls back to the given default.

Parameter	Required	Description
`key`	✔	Description of the parameter.
`default-Value`		A fallback if there are no value for the given key.

Table 6.24. Parameters of cm.localParameter

```
<#assign booleanExample=cm.localParameter("parameterName", false)/>
```

Example 6.14. Returns a single parameter from the localParameters map.

`cm.localParameters()`

Returns a map of all parameters set in a previous template.

```
<#-- all parameters: -->
<#assign examples=cm.localParameters() />
```

```
<#-- single parameters: -->
<#assign booleanExample=cm.localParameters().parameterName!false />
```

Example 6.15. Returns the localParameters as map.

`cm.substitute(id, [original], [default])`

Fetching an action state (id) from an action object (original) and substitutes a bean. If the substitution result is null, it will fall back to default, which is `cm.UNDEFINED` by default. For more information see Section 5.4, "Content Placeholders" in *Content Application Developer Manual*.

Parameter	Required	Description
id	✔	The substitution id.
original		The original bean.
default		Optional fallback bean. Default is `UNDEFINED`.

Table 6.25. Parameters of substitute

```
<<#-- @ftlvariable name="self" type="com.mycompany.Action" -->
<#assign substitutionID="example" />
<@cm.include self=cm.substitute(substitutionID, self) />
```

Example 6.16. Use of cm.substitute().

Utilities

`cm.message`

Translates a message key into a localized message based on `java.text.MessageFormat`. This output is not escaped by default.

Parameter	Required	Description
key	✔	Translates a message key into a localized message via Spring Framework.
args		Additional parameter as Array to enrich the output with functionality.
escaping		Additional Boolean parameter for escaping, default value is **false**.

Parameter	Required	Description
highlightErrors		Specifies if errors should be highlighted, default value is **true**.

Table 6.26. Parameters of message

```
<button class="btn-close"><@cm.message "button_close"/></button>
```

Example 6.17. Renders a localized button with the given key "button_close"

`cm.getMessage(key, [args], [highlightErrors])`

Translates a message key into a localized message based on `java.text.MessageFormat`. Use `?no_esc` to avoid escaping, if the message includes HTML.

Parameter	Required	Description
key	✔	Translates a message key into a localized message.
args		Additional parameter to enrich the output with functionality.
highlightErrors		Specifies if errors should be highlighted, default value is **false**.

Table 6.27. Parameters of getMessage

```
<button class="btn-close" title="${cm.getMessage("button_close")}">X</button>
```

Example 6.18. Renders a button with localized title

```
<#assign messageArgs=[5, "Hello World"] />
<div title="${cm.getMessage("search_results", messageArgs)}">
  <@cm.message key="search_results" args=messageArgs />
</div>
```

Example 6.19. Example of `cm.message` and `cm.getMessage()` with arguments

`cm.hasMessage(key)`

Checks if a translation for a given key exists.

Parameter	Required	Description
key	✔	Checks if a message key exists, if no message found it will return an empty String.

Table 6.28. Parameter of hasMessage

```
<#assign titleKey=fragmentView.titleKey!""/>
<#if titleKey?has_content && (cm.hasMessage(titleKey))>
    <@cm.message titleKey/>
</#if>
```

Example 6.20. Checks if a translation for a message exists and translates the message key into a localized String.

6.5.2 Preview (preview)

The preview FreeMarker API provides calls to render inline metadata information about content, prints out additional script sources for a CAE preview in Studio and supports specific Boolean calls. It uses the namespace `preview` for template calls.

Metadata

`preview.metadata`

Provides inline metadata information to be used for the CAE. This metadata is used by *Studio*. For more information see Section 4.3.5, "Adding Document Metadata" in *Content Application Developer Manual*.

Parameter	Required	Description
data	✔	Prints serialized metadata.

Table 6.29. Parameter of metadata

```
<div<@preview.metadata self.content/>>
  <h1<@preview.metadata "properties.title"/>>${self.title}</h1>
```

```
  <div<@preview.metadata "properties.text"/>>${self.text}</div>
</div>
```

Example 6.21. Getting Metadata for a container with title and text.

`preview.previewScripts`

Prints all scripts and styles necessary for handling the `preview.metadata` by CAE to *Studio*. Should be added in the HTML head.

```
<@preview.previewScripts/>
```

Preview CAE Checks

`preview.isPreviewCae()`

Returns true if *CAE* is running as *Preview CAE*.

```
<#if preview.isPreviewCae()>...</#if>
```

`preview.isFragmentPreview()`

Returns true if *CAE* is rendering a fragmented preview of a content.

```
<#if preview.isFragmentPreview()>...</#if>
```

`preview.getStudioAdditionalFilesMetadata(cssList, jsList)`

Returns optional serialized metadata files in the header in order to render additional Studio specific CSS and JS in the preview frame. For more information see Section 4.3.5, "Adding Document Metadata" in *Content Application Developer Manual*.

Parameter	Required	Description
cssList	✔	Gets CSS sources for the Studio preview.
jsList	✔	Gets JavaScript sources for the Studio preview.

Table 6.30. Parameters of getStudioAdditionalFilesMetadata

```
<#assign studioExtraFiles=preview.getStudioAdditionalFilesMetadata(
    bp.setting(self, "studioPreviewCss"),
```

```
    bp.setting(self, "studioPreviewJs")
)/>

<head<@preview.metadata data=studioExtraFiles/>>
  ...
</head>
```

Example 6.22. Include CSS and JavaScript from content settings with the names "studioPreviewCss" and studioPreviewJs".

`preview.content`

Function to get the content information of a given object that can be used to render preview information. If no content information was found, `cm.UNDEFINED` is returned (see Section 6.5.1, “CoreMedia (cm)” [175]).

```
<@preview.metadata data=[preview.content(self)] />
```

6.5.3 Blueprint (bp)

The Blueprint FreeMarker API provides calls for blueprint specific functionality like settings, markup, images, and localization. It also includes some typical helper utilities like ids, buttons and more. It uses the namespace `bp` for template calls.

Core

`bp.isActiveNavigation(navigation, navigationPathList)`

Returns true if the given `navigation` object is contained in the `navigationPathList`.

Parameter	Required	Description
navigation	✔	Navigation object
navigationPathList	✔	List of navigation objects to check

Table 6.31. Parameters of isActiveNavigation

```
<#if (bp.isActiveNavigation(self, (cmpage.navigation.navigationPathList)![]))>
```

```
  <#assign cssClass=cssClass + ' active'/>
</#if>
```

Example 6.23. Assign a CSS class if this element is part of the navigation list.

`bp.setting(self, key, [default])`

Returns a setting for a given `key` or the `default` value. The lookup for the given key will first check the given ContentBean `self`, secondly the `context`, like the Page and finally the theme.

Parameter	Required	Description
`self`	✔	Settings object.
`key`	✔	Key for the wanted setting.
`default`		Possible default value.

Table 6.32. Parameters of setting

```
<#assign maxDepth=bp.setting(self, "navigation_depth", 2) />
```

Example 6.24. Define a `"maxDepth"` setting or default to `2`.

`bp.generateId([prefix])`

Generates a unique HTML element id with the given prefix.

Parameter	Required	Description
`prefix`		The prefix to add to the id.

Table 6.33. Parameters of generateId

```
<#assign formId=bp.generateId('example') />

<label for="${formId}">Label</label>
<input id="${formId}" type="text" name="example">
```

Example 6.25. Generate an ID for a form input.

`bp.truncateText(text, [maxLength])`

Shortens a `text` at the first space character after `maxLength`.

Parameter	Required	Description
`text`	✔	Text to be truncated.
`maxLength`		Text length limit based on characters.

Table 6.34. Parameters of truncateText

```
<@bp.truncateText(self.teaserText!"", bp.setting(cmpage, "text.max.length",
 200)) />
```

Example 6.26. Shorten a teaser text to a limit, defined in the page settings or default to 200.

`bp.truncateHighlightedText(text, [maxLength])`

Same as `bp.truncateText(text, maxLength)`, but it will keep highlighted elements. Used in search result pages.

Parameter	Required	Description
`text`	✔	Text to be truncated.
`maxLength`		Text length limit based on characters.

Table 6.35. Parameters of truncateHighlightedText

`bp.isEmptyRichtext(richtext)`

Checks if the given richtext is empty without the richtext grammar.

Parameter	Required	Description
`richtext`	✔	The richtext to be checked.

Table 6.36. Parameters of isEmptyRichtext

```
<#if !bp.isEmptyRichtext(self.teaserText!"")>
  <div class="cm-teaser__text">
    <@cm.include self=self.teaserText />
  </div>
</#if>
```

Example 6.27. Check if the teaserText is empty.

`bp.previewTypes(page, self, [defaultFragmentViews])`

Returns the preview views of an object based on its hierarchy as a list.

Parameter	Required	Description
self	✔	The object to preview.
page	✔	The page used to find the setting named "fragmentPreview".
defaultFragmentViews		A Map defining defaults.

Table 6.37. Parameters of previewTypes

`bp.getStackTraceAsString(exception)`

Returns a string including the whole Java stack trace of an exception.

Parameter	Required	Description
exception	✔	Exception of which to return the stack trace.

Table 6.38. Parameters of getStackTraceAsString

```
<textarea class="stacktrace">${bp.getStackTraceAsString(self)!""}</textarea>
```

Example 6.28. Assign the link to this CMVideo object to a variable.

`bp.isWebflowRequest`

Checks, if this current request is a Spring Web Flow request.

```
<#assign isWebflowRequest=bp.isWebflowRequest()/>
<#assign fragmentLink=cm.getLink(self.delegate, "fragment", {
  "targetView": self.view!cm.UNDEFINED,
  "webflow": isWebflowRequest
})/>
```

Example 6.29. Assign the link to this CMVideo object to a variable.

`bp.getDisplayFileSize(size, locale)`

Returns the entered `size` in human readable format.

Parameter	Required	Description
size	✔	Size as integer.
locale		Optional locale. If not set, the locale of the context (page) is used, if available. Fallbacks to the locale of the `RequestContext`.

Table 6.39. Parameters of getDisplayFileSize

`bp.getDisplayFileFormat(mimeType)`

Returns the file extension for a given `mimeType`. For example "image/jpeg" would return "jpg".

Parameter	Required	Description
mimeType	✔	Mime type to translate into its file extension.

Table 6.40. Parameters of getDisplayFileFormat

`bp.isDisplayableImage(blob)`

Checks if this `blob` is of the mime type "image".

Parameter	Required	Description
blob	✔	Blob to be checked.

Table 6.41. Parameters of isDisplayableImage

```
<#if self.blob?has_content && bp.isDisplayableImage(self.blob)>
  ...
```

Example 6.30. Check if this blob has content and is an image.

`bp.isDisplayableVideo(blob)`

Checks if this `blob` is of the mime type "video".

Parameter	Required	Description
`blob`	✔	Blob to be checked.

Table 6.42. Parameters of isDisplayableVideo

```
<#if self.blob?has_content && bp.isDisplayableImage(self.blob)>
  ...
```

Example 6.31. Check if this blob has content and is a video.

`bp.getLinkToThemeResource(path)`

Retrieves the URL path that belongs to a theme resource (image, web font, etc.) defined by its path within the theme folder. The path must not contain any descending path segments.

Parameter	Required	Description
`path`	✔	Path to the resource within the theme folder.

Table 6.43. Parameters of getLinkToThemeResource

```
<img src="${bp.getLinkToThemeResource('img/logo.jpg')}"/>
```

Example 6.32. Using the path to an image.

See Section 5.7, “Referencing a Static Theme Resource in FreeMarker” [76] to learn more about referencing static theme resources.

Grid

`bp.getPageMetadata(page)`

Returns the first navigation context within the navigation hierarchy.

Parameter	Required	Description
page	✔	The page metadata of content.

Table 6.44. Parameter of getPageMetadata

```
<html <@preview.metadata data=bp.getPageMetadata(self)!""/>>
  <@cm.include self=self view="_head"/>
  <@cm.include self=self view="_body"/>
</html>
```

Example 6.33. Renders metadata information to the HTML tag

`bp.getPlacementPropertyName(placement)`

Returns the name of a given placement.

Parameter	Required	Description
placement	✔	Returns the property name of a given PageGridPlacement or `""`.

Table 6.45. Parameter of getPlacementPropertyName

```
<#-- This placement is used for the footer section -->
<footer id="cm-${self.name!""}" class="cm-footer"<@preview.metadata
[bp.getPlacementPropertyName(self)!"",

bp.getPlacementHighlightingMetaData(self)!""]/>>
  ...
</footer>
```

Example 6.34. Renders the placement name to the metadata section.

`bp.getContainer(items)`

Utility function to allow rendering of containers with custom items, for example, partial containers with an item subset of the original container.

Parameter	Required	Description
item	✔	The items to be put inside the new container. Returns a new container.

Table 6.46. Parameter of getContainer

```
<#if self.related?has_content>
  <@cm.include self=bp.getContainer(self.related) view="related"/>
</#if>
```

Example 6.35. Gets the container for a related view.

`bp.getDynamizableContainer(object, propertyPath)`

Utility function to render possibly dynamic containers. A dynamic container will be rendered for dynamic inclusion by caching infrastructure (ESI) or the client (AJAX). The decision, if a container is dynamic or not, is performed on the server side via DynamicContainerStrategy implementations and does not require any further client-side or template logic.

Parameter	Required	Description
object	✔	The object backing the dynamizable container. Can be a content bean producing a list of beans that may contain dynamic items, such as a personalized content bean.
propertyPath	✔	A possible nested property path referencing the list of beans for inclusion. Example: If object is an instance of CMTeasable the property path 'related' references the teasable's related items.

Table 6.47. Parameter of getDynamizableContainer

`bp.getContainerFromBase(baseContainer, [items])`

Utility function to allow rendering of containers with custom items, for example partial containers with an item subset of the original container.

Parameter	Required	Description
baseContainer	✔	The base container from which the new container should be created.
items		The items to be put inside the new container.

Table 6.48. Parameters of getContainerFromBase

```
<@cm.include self=bp.getContainer(self.media)
             view="asTeaser"/>
```

Example 6.36. A new container is created with a new subset of items and rendered as a teaser

`bp.getPageLanguageTag(object)`

Renders the value of the `lang` attribute for the `HTML` tag.

Parameter	Required	Description
object	✔	Object to determine the locale from IETF BCP 47 language code.

Table 6.49. Parameter of getPageLanguageTag

```
<!DOCTYPE html>
<html lang="${bp.getPageLanguageTag(cmpage!self)}">
...
</html>
```

Example 6.37. Renders the value of the lang attribute.

`bp.getPageDirection(object)`

Renders the value of the `dir` attribute for the `HTML` tag according to the locale of the page.

Parameter	Required	Description
`object`	✔	Object to determine the locale direction from `"ltr"` or `"rtl"`.

Table 6.50. Parameter of getPageDirection

```
<!DOCTYPE html>
<html dir="${bp.getPageDirection(cmpage!self)!'ltr'}">
...
</html>
```

Example 6.38. Renders the value of the dir attribute.

`bp.getPlacementHighlightingMetaData(placement)`

Returns a map which contains information about the state of the given placement. The map contains information about the name, and if it is in the layout and if it has items. These metadata information are used by the Studio Preview, see Section 2.4.1, “Content App” in *Studio User Manual*

Parameter	Required	Description
`placement`	✔	The placement of a pagegrid to get the information for.

Table 6.51. Parameter of getPlacementHighlightingMetaData

```
<div <@preview.metadata
data=[bp.getPlacementHighlightingMetaData(pagrid.placement)!""]/>>
  ...
</div>
```

Example 6.39. Renders a div with additional data attribute containing information about the state of the placement.

Images

`bp.responsiveImageLinksData(picture, [aspectRatios])`

Adds responsive relevant image data as additional attribute to a picture.

Parameter	Required	Description
picture	✔	The given image.
aspectRatios		List of aspect ratios to use for this image.

Table 6.52. Parameters of responsiveImageLinksData

```
<#if self.data?has_content>
  <#assign classResponsive="cm-media--responsive"/>
  <#assign attributes += {"data-cm-responsive-media":
bp.responsiveImageLinksData(self)!""}/>

  <img src="#" ${classResponsive!""}" <@bp.renderAttr attributes/>
</#if>
```

Example 6.40. Adding responsive attribute data to an image

`bp.getBiggestImageLink(picture, aspectRatio, makeAbsolute)`

Returns the image link of the biggest image for a given aspect ratio, defined in the Responsive Image Settings.

Parameter	Required	Description
picture	✔	The CMPicture image for which an URL should be rendered.
aspectRatio		The given aspect ratio, the default value is "".
makeAbsolute		Whether the returned link should be absolute. The default value is `false`.

Table 6.53. Parameters of getBiggestImageLink

```
<#assign fullImageLink=bp.getBiggestImageLink(self, "exampleAspectRatioName")/>
```

```
<a href="${fullImageLink}" title="${self.title!""}"
data-cm-popup="gallery">...</a>
```

Example 6.41. Renders the biggest image link of a page

`bp.transformedImageUrl(picture, aspectRatio, width, height)`

Returns the link for an image in the given aspect ratio, width and height.

Parameter	Required	Description
picture	✔	The CMPicture image for which an URL should be rendered.
aspectRatio	✔	The given aspect ratio.
width	✔	The given width in px.
height	✔	The given height in px.

Table 6.54. Parameters of transformedImageUrl

```
<#assign mobileImageUrl=bp.transformedImageUrl(self, "2x3", "200", "300")/>
<img src="${mobileImageUrl}" alt="${self.title!""}" />
```

Example 6.42. Renders a specific size and aspect ratio of an image

6.5.4 LiveContext (lc)

The LiveContext FreeMarker API provides utility functions of the LiveContextFreemarker-Facade to enrich pages with product specific data and components. It uses the namespace `lc` for template calls.

Prices

`lc.formatPrice(amount, currency, locale)`

Formats a given price according to the currency and locale.

Parameter	Required	Description
amount	✔	The numeric part of the price.

Parameter	Required	Description
currency	✔	The currency of the price.
locale	✔	The locale to be used.

Table 6.55. Parameters of formatPrice

```
<#list self.orderItems![] as item>
  <#assign totalPriceFormatted=lc.formatPrice(item.price,
item.product.currency, item.product.locale)/>
  <div>${totalPriceFormatted!""}</div>
</#list>
```

Example 6.43. List all items in a cart with given price

`lc.createProductInSite(product)`

To be used for a product representation in several sites.

Parameter	Required	Description
product	✔	A product representation.

Table 6.56. Parameter of createProductInSite

```
<#list self.orderItems![] as item>
  <#assign productInSite=lc.createProductInSite(item.product)/>
  <a href="${cm.getLink(productInSite)}">${item.product.name!""}</a>
</#list>
```

Example 6.44. List all product links in a cart

`lc.previewMetaData()`

Returns a map containing information for preview of fragments.

`lc.augmentedContent()`

Returns true if the current fragment request targets an Augmented Page.

Name of eCommerce Vendor

`lc.getVendorName()`

Returns name of eCommerce Vendor like `IBM`, `SAP Hybris`, or `coremedia`

User URLs

`lc.getStatusUrl()`

Returns the URL for the status handler to retrieve the actual state (logged in/logged out) of the user.

`lc.getLoginFormUrl()`

Returns the absolute URL to the login form of a commerce system.

`lc.getLogoutUrl()`

Returns the logout URL of a commerce system to logout the current user.

Availability

`lc.availability(product, ifTrue, ifFalse, default)`

Checks if the given product is available. If this is the case the String provided by parameter "ifTrue" will be rendered otherwise the String provided by parameter "ifFalse" will be used. If the availability check cannot be performed (for example, in a fragment preview) the value provided by parameter "default" is rendered.

Please take in mind that the value will be escaped before output. It is currently not possible to pass build-ins like `?no_esc`.

Parameter	Required	Description
product	✔	The `com.coremedia.livecontext.ecommerce.catalog.Product` to check.
ifTrue		The String to be rendered if the product is available. Defaults to `true`.
ifFalse		The String to be rendered if the product is not available. Defaults to `false`.

Parameter	Required	Description
default		The String to be rendered if the availability cannot be checked (for example, in the fragment preview). Defaults to the value of parameter `ifTrue`.

Table 6.57. Parameters of available

```
<div class="cm-product <@lc.availability ifTrue="cm-product--available"
ifFalse="cm-product--not-available" />>
  ...
</div>
```

Example 6.45. Render a CSS class depending on product availability

`lc.createBeanFor(content)`

Generates and returns a content bean for a content from the content type model. Used for pictures.

`lc.createBeansFor(contents)`

Generates and returns a list of content beans for a set of content from its corresponding content type model. Used for visuals and downloads of Products.

6.5.5 Download Portal (am)

The FreeMarker API of the *CoreMedia Advanced Asset Management* for the download portal. It uses the namespace `am` for template calls. For more information see Section 6.6.4.7, “Asset Download Portal” in *Blueprint Developer Manual* .

`am.getDownloadPortal()`

Returns the HTML for the Download Portal.

```
<@cm.include self=am.getDownloadPortal()/>
```

Example 6.46. Render the Download Portal via include

`am.hasDownloadPortal()`

Returns true, if this site contains a Download Portal.

6.5.6 Elastic Social (es)

The Elastic Social FreeMarker API provides utility functions to enrich components with personal data. It uses the namespace `lc` for template calls. For more information see Section 6.3, "Elastic Social" in *Blueprint Developer Manual* .

Complaints

`es.complaining`

Adds user specific data to components and function calls about users which there are complaints. It uses the namespace es for template calls.

Parameter	Required	Description
value	✔	Returns the complain value if true.
id	✔	The HTML id prefix for this component.
collection	✔	The name of collection.
itemId	✔	The name of itemId.
navigationId	✔	The name of navigationId.
customClass		The name of customClass. Defaults to empty.

Table 6.58. Parameters of complaining

```
<@es.complaining id=userDetails.id
                 collection="users"
               value=es.hasComplaintForCurrentUser(userDetails.id, "users")

                 itemId=itemId
                 navigationId=navigationId/>
```

Example 6.47. Enrich user specific data to component

Configuration

`es.getElasticSocialConfiguration(page)`

Gets the Elastic Social configuration of a page. In general this is the root page of a site. Please check the CMS Javadoc for all available properties of `ElasticSocialConfiguration`.

Parameter	Required	Description
page	✔	The page to get the configuration for.

Table 6.59. Parameter of getElasticSocialConfiguration

```
<#assign elasticSocialConfiguration=es.getElasticSocialConfiguration(cmpage)/>
<#if elasticSocialConfiguration.isFeedbackEnabled()!false>
...
</#if>
```

Example 6.48. Checks if Elastic Social is enabled

Login

`es.getLogin()`

Checks page setting for Elastic Social Webflow login form.

```
<@cm.include self=es.getLogin()!cm.UNDEFINED view="asButtonGroup"/>
```

`es.isAnonymousUser()`

Checks if the current user of the web page is a logged-in user or it is an anonymous user. Returns to true if the current user is not logged in.

```
<#if es.isAnonymousUser()>...</#if>
```

`es.isAnonymous(communityUser)`

Checks if the user choose not to publish its user name, profile image, and other personal information with its contributions. Returns to true if the user wants to remain anonymous.

Parameter	Required	Description
communityUser	✔	The user to be checked.

Table 6.60. Parameter of isAnonymous

```
<#if es.isAnonymous(self.author)>...</#if>
```

`es.getCurrentTenant()`

Returns the tenant of the current Thread. Throws Tenant Exception when no tenant has been set.

Tenant information

```
<#assign tenant=es.getCurrentTenant()/>
<#assign myUrl=cm.getLink('/signin/example_' + tenant)/>
<form action="${myUrl!""}" method="post">
  ...
</form>
```

Example 6.49. Sets the form action

`es.hasUserWrittenReview(target)`

Returns the written review of the user for a given bean.

Reviews

Parameter	Required	Description
target	✔	The given bean.

Table 6.61. Parameter of hasUserWrittenReview

`es.getReviewView(review)`

Returns the preview or live rendering depending on the state of the current user.

Parameter	Required	Description
review	✔	Attributing a target with text, title and rating from an author.

Table 6.62. Parameter of getReviewView

```
<#assign reviewView=es.getReviewView(self)/>
<#if ["default", "undecided", "rejected"]?seq_contains(reviewView)>
  ...
</#if>
```

Example 6.50. Specified value rendering

`es.hasUserRated(target)`

Returns the rating score for the given community user and for a given bean. *Rating*

Parameter	Required	Description
target	✔	The given bean.

Table 6.63. Parameter of hasUserRated

`es.getCommentView(comment)`

Returns the preview or live rendering depending on the state of the current user.

Parameter	Required	Description
comment	✔	Attributing a target with text from an author.

Table 6.64. Parameter of getCommentView

```
<#assign commentView=es.getCommentView(self)/>
<#if ["default", "undecided", "rejected"]?seq_contains(commentView)>
  ...
</#if>
```

Example 6.51. Specified value rendering

`es.getMaxRating()`

Returns 5.

`es.getReviewMaxRating()`

Returns 5.

6.5.7 Spring (spring)

The Spring FreeMarker API consists of a collection of FreeMarker macros aimed at easing some of the common requirements of web applications - in particular handling of forms.

For more information see the official Spring documentation. It uses the namespace `spring` for template calls.

6.6 Scripts

The Blueprint Frontend Workspace is a multi-package repository. To keep it simple and fast it includes a lot of tools and scripts. This section describes the available scripts.

Available Scripts

6.6.1 Global Scripts

The following scripts are available in the root folder of the frontend workspace and trigger tasks in the themes and bricks if available.

`pnpm test`

This command will run the `test` script in all available bricks, themes and tools.

`pnpm build`

This command will run the `build` script in all available themes and will create a production build of all the themes.

`pnpm build-frontend-zip`

This command will build a single zip file containing all built themes in `target/frontend.zip`. You need to build the themes before running this script, otherwise the zip file will be empty.

`pnpm run deploy`

This command will run the `deploy` script in all available themes. It runs the `build` script before and uploads the themes to the given Studio. Please see Section 5.6, “Importing Themes into the Repository” [73] and Section 6.6.2, “Theme Scripts” [206] for more details.

`pnpm create-theme [name]`

This command will start the interactive tool to create a new theme with the given name as parameter. The creation wizard will ask you the following questions:

- Do you want to derive the theme from another theme?
- Which bricks should be activated?
- Should non-activated bricks be passed as commented out dependencies?

Please check Section 5.1, “Creating a New Theme” [62] for more details.

`pnpm create-brick [name]`

This command will create a new blank and minimal brick with the given name as parameter in the folder `bricks/`. Please check Section 5.2, “Creating a New Brick” [64] for more details.

`pnpm eject`

This command can eject (creates a copy) of any available brick. The wizard will let you select the bricks from a list and will ask for a new name. The ejected bricks will be created in the folder `bricks/`.

`pnpm prettier`

This command will run the code formatter prettier in all themes and bricks.

6.6.2 Theme Scripts

`pnpm build`

This command will run the module bundler `webpack` for the theme. It will create a minimized and transpiled version of the theme as zip file in the folder `target/themes/` for production.

`pnpm run deploy`

This command will run the `build` task to create a theme zip file and uploads it to the `/Themes` folder in the content repository. You need a valid API key, otherwise you

need to login like in the web developer workflow. You also need write access to the `/Themes` folder. Please see Section 5.6, "Importing Themes into the Repository" [73] for more details.

`pnpm start [--remote|--local]`

This command will start the "watch" task of the theme for development. Please see Chapter 3, *Web Development Workflow* [19] for more details.

`pnpm prettier`

This command will run the code formatter prettier for all files inside the folder `src/js/`. The configuration is defined in file `.prettierrc` and `.prettierignore`.

6.6.3 Brick Scripts

Bricks can offer different scripts depending on the purpose of the brick. CoreMedia default and example bricks include the following scripts:

`pnpm test`

This command will run tests if available. Some bricks are using jest for unit tests.

`pnpm prettier`

This command will run the code formatter prettier for all files inside the folder `src/js/`. The configuration is defined in file `.prettierrc` and `.prettierignore`.

6.6.4 Theme Importer

All CoreMedia themes provide a `theme-importer` script providing commands, which may be helpful only when using a remote Content Application Engine. All commands utilize a REST service co-located with Studio. It may be run by executing the command `pnpm run theme-importer [command]` from your theme directory. The following commands are available.

`pnpm theme-importer login [options]`

This command authenticates a Studio user who is member of the group `development`, requests an API key creates an `apikey.txt` file containing the API key as well as an `env.json` file containing the URLs of Studio and optionally of preview and proxy in the config directory of the Frontend Workspace. If the file `env.json` is already existing, it is only being updated.

The API key expires after one day by default. *CoreMedia on-premise platform* customers may customize the expiration time in the `application.properties` of the Studio web application.

The following options may be passed via the command line.

Parameter	Required	Description
`--studioUrl`	✔	The URL of Studio.
`--username, -u`	✔	A user who is member of the group `development`
`--password, -p`	✔	The password of the user.
`--previewUrl`		The URL of the Studio preview.
`--proxyUrl`		The URL of the proxy server.

Table 6.65. Command-line options for the login command

If required options are not passed as command-line options, they will be prompted for. This way the command may be run without providing any command-line options. The options will all be inquired.

`pnpm theme-importer logout`

This command performs a logout of the user and removes the `apikey.txt` file.

`pnpm theme-importer whoami`

This command outputs information about the logged in user.

`pnpm theme-importer upload-theme`

This command builds the theme and uploads it to the remote Content Application Engine. All files of the theme in the home directory of the logged in developer are being cleared and replaced by the files contained in the recently uploaded theme zip.

If the user is not logged in when running this command, he will be forwarded to the login command.

Glossary

Brick	A reusable frontend package that can contain templates, JavaScript, SCSS/CSS and resource bundles. See Section 6.1, “Example Themes” [103].
browserslist	Library to share target browsers between different frontend tools. See https://github.com/ai/browserslist/
CSS	CSS stands for Cascading Style Sheets and is a style sheet language used to describe the presentation of a document written in HTML.
ECMAScript	Trademarked scripting-language specification standardized by Ecma International in ECMA-262. One of the best-known implementation of ECMAScript is JavaScript.
JavaScript	Interpreted programming language which is one of the three core technologies of web development.
Node.js	Node.js is an open-source, cross-platform JavaScript run-time environment for executing JavaScript code server-side.
npm	npm stands for "Node Package Manager" and is the default package manager for Node.js.
package.json	Contains meta data about an app or module such as its name, version and dependencies. See official Specification.
pnpm	pnpm is an alternative package manager for Node.js.
Prettier	Prettier is a code formatter supporting many languages and integrates with most editors.
Sass	Sass stands for "syntactically awesome stylesheets" and is a scripting language that is interpreted or compiler into CSS.
SCSS	SCSS is a newer syntax for Sass that uses block formatting like CSS.
Theme	In the context of the Frontend Workspace a theme stands for a frontend package that composes templates, JavaScript, SCSS/CSS and resource bundles provided from bricks and third party libraries into a bundle that can be used by the *CAE*. See Section 6.1, “Example Themes” [103].

Webpack	Webpack is an Open-source JavaScript module bundler that is highly extensible by the use of loaders to provide additional tasks and transformations for different file types.
yarn	yarn is an alternative package manager for Node.js.

Index

W